Modelltransformationen als Mittel der modellbasierten Entwicklung von Software-Systemen

Frank Marschall

Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching
Germany

Institut für Informatik
der Technischen Universität München

Modelltransformationen als Mittel der modellbasierten Entwicklung von Software-Systemen

Frank Marschall

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Dr. h.c. Wilfried Brauer

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Dr. Florian Matthes

Die Dissertation wurde am 28.06.2004 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 26.01.2005 angenommen.

Zusammenfassung

Im Software Engineering werden heutzutage verstärkt modellbasierte Entwicklungsansätze eingesetzt, in denen konzeptuelle Modelle mit präzise definierter Semantik für die Beschreibung von Software-Systemen verwendet werden. Durch die Beschränkung auf wesentliche Merkmale und die Möglichkeit die Struktur der Modelle und Modelloperationen in geeigneter Weise zu beschränken tragen diese Ansätze maßgeblich zur Steigerung der Effizienz des Entwicklungsprozesses und der Qualität der erstellten Software-Produkte bei.

Die Bearbeitung konzeptueller Modelle erfolgt in der Regel durch geeignete Beschreibungstechniken, anhand derer sich Sichten auf ein konzeptuelles Modell bilden lassen. Im Rahmen dieser Arbeit werden Techniken erarbeitet, mit denen sich Beschreibungstechniken durch Transformationen ihrer abstrakten Syntax in ein einheitliches konzeptuelles Modell integrieren lassen.

Oftmals werden im Verlauf eines Software-Entwicklungsprozesses konzeptuelle Modelle eines Systems auf verschiedenen Abstraktionsebenen erstellt, die im Idealfall in Teilen durch automatisierte Verfeinerungsabbildungen auseinander hervorgehen. Hierzu werden Verfahren entwickelt, die eine konsistente Verfeinerung und Erweiterung konzeptueller Modelle ermöglichen.

Ein klar definierter Entwicklungsprozess bietet Entwicklern zudem die nötige Hilfestellung bei der Spezifikation und Erstellung eines Software-Systems. Zu diesem Zweck wird ein Framework für die Definition flexibler Entwicklungsprozesse geschaffen, in dem Entwicklungsschritte als Transformationen eines Modells der im Entwicklungsprozess erstellten Artefakte definiert werden.

Die Beschaffenheit korrekter Modelle wird durch Metamodelle definiert. Bedingt durch ihre einfache Abbildbarkeit auf eine Werkzeugunterstützung sind im Software Engineering vor allem objektorientierte Metamodelle weit verbreitet. Um die erarbeiteten Konzepte zur modellbasierten Entwicklung umsetzen zu können, wird die „Bidirectional Object-Oriented Transformation Language“ (BOTL) als geeigneter Mechanismus für die Transformation objektorientierter Modelle entwickelt. Die Sprache verfügt über eine formale Semantik und bietet eine Reihe von Verifikationstechniken zum Nachweis der Ausführbarkeit von Transformationsspezifikationen und der syntaktischen Korrektheit der erzeugten Modelle.

Der Einsatz von BOTL für die Realisierung eines modellbasierten Entwicklungsansatzes wird dargestellt und ihre Verwendung für die Umsetzung der verschiedenen Techniken exemplarisch anhand der KOGITO-Methodik für das Requirements Engineering demonstriert.

Die Realisierbarkeit der erarbeiteten Ansätze zur Modelltransformation in der Praxis wird anhand eines im Rahmen der Arbeit entstandenen Werkzeuges belegt. Das Werkzeug erlaubt die graphische Spezifikation von BOTL-Transformationen und deren automatisierte Verifikation. Eine Transformationskomponente ist schließlich in der Lage objektorientierte Modelle in unterschiedlichen Darstellungen, wie Java-Objektgeflechte oder XMI, gemäß einer BOTL-Spezifikation zu transformieren.

Danksagung

An erster Stelle möchte ich mich bei meinem Doktorvater Herrn Prof. Broy und allen Mitarbeitern seines Lehrstuhls für das angenehme Umfeld, in dem diese Arbeit entstand, bedanken. Herrn Prof. Broy möchte ich insbesonders für die Betreuung und Unterstützung, die mir bei der Erstellung dieser Arbeit zuteil wurde, bedanken. Mein Dank gilt auch Herrn Prof. Matthes, der sich bereit erklärte, das Zweitgutachten für meine Arbeit zu erstellen.

Besonders hervorheben möchte ich an dieser Stelle alle die Kollegen mit denen ich in den Projekten FORSOFT A3, Marlin, ZEN und Kogito im Laufe meiner Zeit am Lehrstuhl zusammenarbeiten durfte. Für die zahlreichen fruchtbaren Diskussionen, ohne die diese Arbeit nicht möglich gewesen wäre, danke ich besonders Gerhard Popp, Wolfgang Prenniger und Maurice Schoenmakers und Peter Braun, mit dem zusammen ich auf dem Gebiet der Modelltransformationen arbeitete.

Mein ganz besonderer Dank gilt jedoch allen meinen Freunden, meiner langjährigen Freundin Theresa sowie meiner Familie für die Geduld und Nachsicht, die sie in der Phase meiner Dissertation für mich aufbrachten. Ohne den Rückhalt, den sie mir gegeben haben, hätte diese Arbeit niemals entstehen können.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Motivation und Überblick . . . 1
1.1.1 Modellierung von Software-Systemen . . . 1
1.1.2 Prozessmodelle im Software-Engineering . . . 3
1.2 Problemstellung . . . 4
1.2.1 Modellierung von Software-Systemen . . . 4
1.2.2 Modellierung von Entwicklungsprozessen . . . 5
1.2.3 Zusammenfassung der Problemstellung . . . 6
1.3 Verwandte Arbeiten . . . 6
1.3.1 Meta Object Facility (MOF) . . . 6
1.3.2 Die Unified Modelling Language (UML) . . . 9
1.3.3 Model Driven Architecture (MDA) . . . 9
1.3.4 MOF 2.0 Query / Views /Transformations (QVT) . . . 11
1.3.5 Graphgrammatiken . . . 13
1.4 Ziele und Lösungsansatz der Arbeit . . . 16
1.4.1 Ziele . . . 16
1.4.2 Inhalt . . . 17
1.4.3 Beitrag der Arbeit . . . 19
1.5 Aufbau der Arbeit . . . 20

2 Grundlagen **23**

2.1 Modellbasierte Software-Entwicklung . . . 23
2.1.1 Modelle und Metamodelle . . . 23
2.1.2 Prozessmodelle . . . 25
2.1.3 Modelle zur Beschreibung von Software-Systemen . . . 28
2.2 Modelltransformationen in der Software-Entwicklung . . . 29
2.2.1 Klassifikation von Spezifikationssprachen für Modelltransformationen . . . 30
2.2.2 Der Prozessmusteransatz . . . 33
2.2.3 Transformation von konzeptuellen Modellen . . . 38
2.2.4 Integration von Artefakten in ein konzeptuelles Modell . . . 42

3 Die Bidirectional Object Oriented Transformation Language (BOTL) **51**

3.1 Objektorientierte Modelle und Metamodelle . . . 51
3.1.1 Primitive Typen und Identifikatoren . . . 51
3.1.2 Objektorientierte Metamodelle . . . 52
3.1.3 Objektorientierte Modelle . . . 58
3.2 Instanzierbarkeit von Metamodellen . . . 63
3.3 BOTL Regelwerke . . . 70

3.4 Ein UML-Profil für BOTL-Regelwerke . . . 78
3.5 Regelwerksanwendung . . . 82
3.5.1 Auffinden von Matches in Quellmodellen . . . 82
3.5.2 Transformation von Quellmodellfragmenten in Zielmodellfragmente . . . 85
3.5.3 Zusammenführen der erzeugten Modellfragmente . . . 92
3.5.4 Regel- und Regelwerksanwendung . . . 98
3.6 Rewrite-Transformationen . . . 102

4 Eigenschaften von BOTL-Regelwerken 109
4.1 Anwendbarkeit von Regelwerken . . . 110
4.1.1 Erzeugen gültiger Modellfragmente . . . 111
4.1.2 Deterministische Objektvariable . . . 115
4.1.3 Konfliktfreie Objektvariable . . . 119
4.1.4 Anwendbarkeit von Regeln und Regelwerken . . . 130
4.2 Metamodellkonformität von Regelwerken . . . 131
4.2.1 Grundlagen . . . 131
4.2.2 Verifikationstechniken für Obergrenzenkonformität . . . 133
4.2.3 Verifikationstechniken für Untergrenzenkonformität . . . 156
4.2.4 Verifikationstechnik für Metamodellkonformität . . . 166
4.2.5 Richtlinien zum Erstellen von BOTL-Spezifikationen . . . 166
4.3 Bijektivität . . . 167
4.4 Metamodellkonformität von Rewrite-Regelwerken . . . 171
4.5 Semantik von Transformationsregeln . . . 173
4.6 Einordnung der BOTL . . . 175

5 Anwendung der BOTL in einem modellbasierten Entwicklungsprozess 179
5.1 Die KOGITO-Methodik . . . 179
5.1.1 Umfeld und Verwandte Ansätze . . . 179
5.1.2 Die konzeptuellen KOGITO-Metamodelle . . . 182
5.1.3 KOGITO-Entwicklungsartefakte . . . 188
5.2 Transformation objektorientierter konzeptueller Modelle mit BOTL . . . 191
5.2.1 Grundsätzliches . . . 193
5.2.2 Eine Abstraktionsabbildung für Requirements Analysis Modelle . . . 194
5.2.3 Eine Verfeinerungsabbildung für Business Requirements Modelle . . . 198
5.3 Integration von Artefakten in ein konzeptuelles Modell . . . 201
5.3.1 Integration von Business E/R-Diagrammen . . . 202
5.3.2 Integration mehrerer Beschreibungstechniken . . . 209
5.4 BOTL-Transformation zur Spezifikation von Vorgehensschritten . . . 216
5.4.1 Spezifikation von Vorgehensschritten . . . 216
5.4.2 Konsistenz von elementaren Aktivitäten und realisierenden Prozessmustern . . . 222
5.4.3 Konsistenz zwischen komplexen und elementaren Prozessmustern . . . 225
5.4.4 Zusammenfassung . . . 230

6 Werkzeugunterstützung für BOTL-Transformationen 231
6.1 Anforderungen und Überblick über das Werkzeug . . . 231
6.1.1 Anforderungen an die Werkzeugunterstützung . . . 231
6.1.2 Prinzipielle Funktionsweise und Grobarchitektur . . . 233

6.2 Datenstrukturen . . . 235
6.2.1 BOTL-Metamodelle . . . 235
6.2.2 BOTL-Regelwerke . . . 237
6.2.3 BOTL-Modelle . . . 239
6.3 Der BOTL-Editor . . . 240
6.3.1 Funktionsumfang des Editors . . . 240
6.3.2 Anbindung der BOTL-Modelle an ArgoUML . . . 243
6.4 Verifikation von Regelwerken . . . 244
6.4.1 Verifikation der Anwendbarkeit von Regelwerken . . . 245
6.4.2 Verifikation der Metamodellkonformität von Regelwerken . . . 247
6.4.3 Erstellung eines Verifikationsportokolls und Darstellung von Fehlern . . . 249
6.5 Die Transformationskomponente . . . 251
6.5.1 Lösen von Gleichungen und Gleichungssystemen . . . 254
6.5.2 Import von Java-Objektgeflechten . . . 256
6.5.3 Auffinden gültiger Matches . . . 257
6.5.4 Transformation von Modellfragmenten . . . 259
6.5.5 Einfügen neuer Fragmente in das Zielmodell . . . 261
6.5.6 Export von Java-Objektgeflechten . . . 261
6.5.7 Verwendung des Transformators . . . 262
6.6 Verwandte Werkzeuge und Ansätze . . . 262
6.6.1 PROGRES . . . 262
6.6.2 OPTIMIX und GREAT . . . 267
6.6.3 GReAT . . . 267
6.6.4 Generative Model Transformer (GMT) und UMLX . . . 270
6.6.5 Zusammenfassung . . . 272

7 Zusammenfassung und Ausblick 275
7.1 Zusammenfassung . . . 275
7.2 Ausblick . . . 276

A Beweise 279
A.1 Beweise zum BOTL-Formalismus . . . 279
A.1.1 Instanzierbarkeit von Metamodellen . . . 279
A.1.2 Regelwerksanwendung . . . 280
A.1.3 Rewrite-Transformationen . . . 286
A.2 Beweise zu Eigenschaften von BOTL-Regelwerken . . . 287
A.2.1 Anwendbarkeit . . . 287
A.2.2 Metamodellkonformität . . . 294
A.2.3 Bijektivität . . . 307
A.2.4 Modell Rewriting-Transformationen . . . 309

B Metamodellkonformität des Regelwerks zur Integration von E/R-Diagrammen 315
B.1 Anwendbarkeit des Regelwerks . . . 315
B.1.1 Anwendbarkeit von r_1 . . . 315
B.1.2 Anwendbarkeit von r_2 . . . 316
B.1.3 Anwendbarkeit von r_3 . . . 318
B.1.4 Anwendbarkeit von r_4 . . . 321

B.1.5 Anwendbarkeit von r_5 321
B.1.6 Anwendbarkeit von r_6 323
B.1.7 Anwendbarkeit von R_{ER} 324
B.2 Obergrenzenkonformität 325
B.3 Untergrenzenkonformität 336

C Das Regelwerk R_{AD} 339

D Beispiel zu Abschnitt 5.3.2 345

E Automatisch generierter Nachweis der Metamodellkonformität 347

Literaturverzeichnis 351

Index 360

Abbildungsverzeichnis

1.1 Die vier Metaebenen der MOF . . . 7
1.2 Exemplarische Abbildung eines plattformunabhängigen Modells auf ein plattformspezifisches Modell . . . 11
1.3 Beispiel für eine Graphgrammatik . . . 14
1.4 Entstehung eines Modells durch Anwendung einer Graphgrammatik . . . 15
1.5 Graphische Darstellung des in Abbildung 1.4 erzeugten Modells als Automat . . . 15

2.1 Zusammenhang zwischen Modell, Metamodell und semantischem Modell . . . 25
2.2 Die MOF Metamebenen und das Metamodell für Entwicklungsprozesse . . . 27
2.3 Modelltransformationen innerhalb eines modellbasierten Entwicklungsprozesses . . . 30
2.4 Zusammenhang zwischen Produktmetamodell und Aktivitätsmetamodell im Prozessmusteransatz . . . 34
2.5 Beispielhafte Darstellung der Beziehungen zwischen Prozessmustern und Aktivitäten 36
2.6 Integration von Artefakten in ein konzeptuelles Modell . . . 44
2.7 Verfeinerung von Modellen und Integration von Artefakten im Rahmen eines MDA-basierten Entwicklungsprozesses . . . 47

3.1 Beispiel für eine Klasse . . . 53
3.2 Die Klassenassoziation AE_1 . . . 55
3.3 Die Klassenassoziation AE_2 . . . 55
3.4 Die symmetrische Klassenassoziation AE_3 . . . 56
3.5 Das Metamodell mm_α . . . 57
3.6 Das Metamodell mm_β . . . 57
3.7 Ein Objekt . . . 59
3.8 Eine Objektassoziation . . . 60
3.9 Eine symmetrische Objektassoziation . . . 61
3.10 Das objektorientierte Modell m_α . . . 62
3.11 Minimalbeispiel für ein *nicht* instanzierbares Metamodell . . . 63
3.12 Das Metamodell *mm* . . . 65
3.13 Die möglichen Konfigurationen der Klasse *A* des Metamodells *mm* . . . 65
3.14 Beispiel für ein instanzierbares Metamodell . . . 68
3.15 Eine gültige Instanz des in Abbildung 3.14 dargestellten Metamodells . . . 70
3.16 Eine Objektvariable . . . 72
3.17 Eine Modellvariable . . . 74
3.18 Die Modelltransformationsregel r_0 . . . 75
3.19 Die Modelltransformationsregel r_1 . . . 77
3.20 Die paramterisierbare Regel r(\$NAME, \$BOOL) . . . 77
3.21 Die durch r(„ACME“, \$BOOL) repräsentierten BOTL-Regeln . . . 78

3.22 Beispiel für die Darstellung eines BOTL-Metamodells mit dem UML-Profil für BOTL 81
3.23 Beispiel für die Darstellung einer BOTL-Modellvariable mit dem UML-Profil für BOTL 81
3.24 Eine alternative Darstellungsform der Modellvariable aus Abbildung Abbildung 3.23 81
3.25 Ein Modellfragment-Match $(mv_0, mf, match_o, match_a)$ 85
3.26 Die Modellfragmente (a) $mfm_{0,0}|_{mf}$ und (b) $mfm_{0,1}|_{mf}$ 85
3.27 Die Regel r_1 mit einem Modellfragment-Match auf der linken Seite und einem Modellfragment mf_1, wobei $mfr(mfm_i^0, r_1, mf_1)$ gilt 88
3.28 Die erzeugten Modellfragmente (a) $mf_{0,0}$ und (b) $mf_{0,1}$ 91
3.29 Die Modellfragmente mf_0 und mf_1 93
3.30 Zwei Modellfragmente und das Ergebnis der $\cup_m$-Operation 96
3.31 Das Ergebnis von $apply(m_\alpha, r_0) = mf_0$ 100
3.32 Das Ergebnis $mf_\beta := transform(\{m_\alpha\}, R$ der Anwendung des Beispielregelwerks R 100
3.33 Das Metamodell mm_0 105
3.34 Das Regelwerk $R^{id}_{mm_0}$ 105

4.1 Eigenschaften von BOTL-Regelwerken 109
4.2 Beispiel für eine Regel die *keine* gültigen Modellfragmente erzeugt. 111
4.3 Die Regel r 114
4.4 Beispiel für eine *nicht* deterministische Objektvariable vom Typ Person 116
4.5 Die Regel r_1, für deren Quellmodellvariable $r_1|_{mv_0}$ gilt: $\{ov0\} \overset{r_1|_{mv_0}}{\rightsquigarrow} \{pv0\}$ 118
4.6 Beispiel für eine Regel mit zwei *nicht* konfliktfreien Objektvariablen $pv0$ und $pv1$ . . 120
4.7 Zwei von der Regel aus Abbildung 4.6 erzeugte Modellfragmente 121
4.8 Regel r_{appl} 123
4.9 Regel r_{nappl} 128
4.10 Eine Objektvariablenassoziationen und die durch sie erzeugten Objektassoziationen . 134
4.11 Ein Beispiel für $OVP(r_{super}, r_{sub})$ 138
4.12 Eines von zwei möglichen Mappings zwischen r_0 und r_1 und die entsprechende Menge OVP_0. 139
4.13 Das zweite mögliche Mapping zwischen r_0 und r_1 und die entsprechende Menge OVP_1.140
4.14 Abhängigkeiten zwischen *maxRelevant*, *maxRedundant* und ihren (pessimistischen) Schätzheuristiken *relevant* und *redundant* 141
4.15 Drei Modelltransformationsregeln 144
4.16 Das gemeinsame Quellmetamodell der drei Regeln 144
4.17 Ein Ausschnitt eines Quellmodells bei dem ov_0, ov_1 und ov_2 jeweils auf das Objekt mit der Identität *anyID* matchen würden. 145
4.18 Graphische Darstellung des Verhältnisses der Mengen die in $\mathbb{OV}_R^{conf}$ und $\mathbb{OV}_R^{conf*}$ enthalten sind. 146
4.19 Das Metamodell *mm* 147
4.20 Die Modellvariable mv_1 148
4.21 Die Modellvariable mv_2 148
4.22 Beispiel für ein Modellfragment für *maxmatch* 148
4.23 (a) Das Quellmetamodell mm_0 (b) Die Quellmodellvariable mv_0 149
4.24 Das einzig mögliche Modellfragment auf das mv_0 bei festem ov_a und ov_b maximal oft matchen kann. 150
4.25 Die maximal vier möglichen Matches von mv_0 bei festem ov_a und ov_b in einem zu mm_0 konformen Modell. 151

4.26 Reflexive Assoziation $(ov_0 = ov_1, ae_0 \neq ae_1)$ . . . 153
4.27 Symmetrische und reflexive Assoziation $(ov_0 = ov_1, ae_0 = ae_1)$ . . . 153
4.28 ov_0 und ov_1 . . . 155
4.29 Summe der Kardinalitäten von ausgehenden Assoziationen . . . 157
4.30 Beispiel für ein Objekt, das von zwei Objektvariablen erzeugt wurde . . . 158
4.31 Reflexive Assoziation $(ov_0 = ov_1, ae_0 \neq ae_1)$ . . . 162
4.32 Symmetrische und reflexive Assoziation $(ov_0 = ov_1, ae_0 = ae_1)$ . . . 162
4.33 ov_0 und ov_1 . . . 162
4.34 Beweiskette für die Verifikationstechnik für Untergrenzenkonformität . . . 165
4.35 Das Metamodell mm_A . . . 173
4.36 Die Regel r . . . 175

5.1 Die in KOGITO verwendeten konzeptuellen Metamodelle . . . 183
5.2 Das Business Reference Metamodell mm_{Ref} . . . 184
5.3 Das Business Requirements Metamodell mm_{BRM} . . . 185
5.4 Das Requirements Analysis Metamodell mm_{RAM} . . . 187
5.5 Das Produktmodell der Entwicklungsartefakte für die Beschreibung des Business Reference Modells . . . 190
5.6 Das Metamodell mm_{ER} zur Spezifikation der abstrakten Syntax von Entity-Relationship-Diagrammen . . . 191
5.7 Abstraktions- und Verfeinerungsabbildung zwischen konzeptuellen KOGITO-Modellen 192
5.8 Beispiel für ein gültiges Business Requirements Modell m_{BRM} . . . 194
5.9 Abbildungsregel r_{a1} . . . 195
5.10 Abbildungsregel r_{a2} . . . 196
5.11 Abbildungsregel r_{a3} . . . 196
5.12 Abbildungsregel r_{a4} . . . 197
5.13 Abbildungsregel r_{a5} . . . 197
5.14 Das durch die Anwendung des Regelwerks R_{BRM} aus dem in Abbildung 5.8 dargestellten Modell erzeugte Requirements Analysis Modell . . . 201
5.15 Integration von Beschreibungstechniken in ein KOGITO Requirements Analysis Modell 202
5.16 Beispiel für ein Entity-Relationship-Diagramm . . . 202
5.17 Instanzmodell m_{ER} der abstrakten Syntax des E/R-Diagramms aus Abbildung 5.16 . 203
5.18 Abbildungsregel r_1 . . . 204
5.19 Abbildungsregel r_2 . . . 204
5.20 Abbildungsregel r_3 . . . 205
5.21 Beispiel für eine Anwendung der Regel r_3 . . . 205
5.22 Abbildungsregel r_4 . . . 206
5.23 Abbildungsregel r_5 . . . 206
5.24 Abbildungsregel r_6 . . . 207
5.25 Die aus dem E/R-Diagramm erzeugte Sicht des konzeptuellen KOGITO-Modells . . 208
5.26 Darstellung des konzeptuellen Datenmodells aus Abbildung 5.25 in Form eines Klassendiagramms . . . 209
5.27 Das Artefakt „Create Order“ vom Typs „Business Activity Diagram“ . . . 210
5.28 Instanzmodell m_{AD} des in Abbildung 5.27 dargestellten Artefakts . . . 211
5.29 Die aus dem Aktivitätsdiagramm „Create Order“ erzeugte Sicht des konzeptuellen KOGITO-Modells . . . 212

5.30 Das durch Zusammenführen der erzeugten Modellfragmente entstandene Requirements Analysis Modell m_{RAM} . . . 213
5.31 Das durch die Anwendung von Prozessmustern entstandene Modell der Artefakte . . 224
5.32 Die Regel id_1 . . . 227
5.33 Die Regel id_2 . . . 227

6.1 Überblick über das BOTL-Werkzeug . . . 233
6.2 Die Abhängigkeiten zwischen den Komponenten des Prototypen . . . 234
6.3 Die Pakete und Klassen für Metamodelle, Modelle und Regelwerke . . . 236
6.4 Oberfläche der ArgoUML-Erweiterung zur Spezifikation von BOTL-Regeln und -Metamodellen . . . 241
6.5 Menu mit den BOTL-Erweiterungen in ArgoUML . . . 241
6.6 Menu mit den BOTL-Erweiterungen für den Demonstrationsmodus . . . 242
6.7 Zusammenhang zwischen den ArgoUML-Modellen und dem BOTL-Modell . . . 243
6.8 Das Paket `errorProcessing` mit den Klassen für die Behandlung von Fehlern während der Verifikation eines Regelwerks . . . 250
6.9 Ausgabe des Protokolls nach der Verifikation eines Regelwerks . . . 251
6.10 Ausgabe von Fehlermeldungen nach der Verifikation eines Regelwerks . . . 252
6.11 Ablauf einer Modelltransformation . . . 253
6.12 Eine Quellmodellvariable die einen Constraint formuliert . . . 258
6.13 Ein Modellfragment das zusammen mit der Quellmodellvariable aus Abbildung 6.12 einen Match bildet. . . . 259
6.14 Beispiel für die interne Darstellung eines Graphen in PROGRES [Sch97] . . . 263
6.15 Beispiel für ein zum Graphen aus Abbildung 6.15 passendes PROGRES-Schema [Sch97] . . . 264
6.16 Beispiel für einen PROGRES-Test [Sch97] . . . 265
6.17 Beispiel für eine PROGRES-Produktion [Sch97] . . . 266
6.18 Eine Produktionsregel im GReAT-Werkzeug . . . 268
6.19 Sequentielle Ausführung von Produktionen . . . 269
6.20 Graphische Elemente von UMLX, um welche die UML erweitert wird [Wil03b] . . . 271
6.21 Beispiel für eine zusammengesetzte UMLX-Transformation [Wil03b] . . . 272

A.1 Die Assoziationsregel $r_i a$ zur Klassenassoziation AE_i . . . 310

C.1 Abbildungsregel r_{AD1} . . . 339
C.2 Abbildungsregel r_{AD2} . . . 340
C.3 Abbildungsregel r_{AD3} . . . 340
C.4 Abbildungsregel r_{AD4} . . . 341
C.5 Abbildungsregel r_{AD5} . . . 341
C.6 Abbildungsregel r_{AD6} . . . 341
C.7 Abbildungsregel r_{AD7} . . . 342
C.8 Abbildungsregel r_{AD8} . . . 343
C.9 Abbildungsregel r_{AD9} . . . 343

D.1 Das Modell m'_{AD} . . . 345
D.2 Ergebnis der Transformation $m'_{RAM} := transform(\{m'_{AD}, (\emptyset, \emptyset)\}, R_{BRM} \cup R_{ER} \cup R_{AD})$ . . . 346
D.3 Ergebnis der Transformation $m''_{BRM} := transform\big(transform(m'_{RAM}, R_a)$ . . . 346

E.1 Die Regel `r1` 347
E.2 Die Regel `r1` 347

Tabellenverzeichnis

3.1 Die UML-Stereotyen des BOTL-Profils 79
3.2 Tagged Values des UML-Profils für BOTL 80

4.1 Berechnung der Summanden für Fall (iii) von *cannotCreateSame* 144
4.2 Definition von *ubVarCard* für den Fall $ov_0 \neq ov_1$ 154
4.3 Definition von *lbVarCard* für den Fall $ov_0 \neq ov_1$ 163
4.4 Semantik für das Metamodell mm_A 174

5.1 Beispiel für ein Artefakt des Typs „Business Process Description Worksheet“ 189
5.2 Konkrete Syntax der in KOGITO verwendeten Entity-Relationship-Digagramme . . 192
5.3 Das Prozessmuster „Reference Model Creation“ 220
5.4 Die Aktivität „Create Business Area“ 221
5.5 Die Aktivität „Create Process Area“ 221
5.6 Das Prozessmuster „Business Area Creation“ 222
5.7 Das Prozessmuster „Process Area Creation“ 223

B.1 Nachweis der Konfliktfreiheit von Objektvariablen aus unterschiedlichen Regeln von R_{ER} 326
B.2 Für den Nachweis der Metamodellkonformität benötigte Bezeichner (1/2) 327
B.3 Für den Nachweis der Metamodellkonformität benötigte Bezeichner (2/2) 328
B.4 Nachweis der Untergrenzenkonformität der einzelnen Regeln 337

Definitionen

Definition 2.1.1 Modell . . . 23
Definition 2.1.2 Metamodell, Metametamodell . . . 24
Definition 2.1.3 Semantik, semantisches Modell . . . 24
Definition 2.1.4 Artefakt . . . 25
Definition 2.1.5 Artefakttyp . . . 25
Definition 2.1.6 Produktmodell, Modell der Artefakte . . . 26
Definition 2.1.7 Aktivitätsbeschreibung . . . 26
Definition 2.1.8 Prozessmodell . . . 26
Definition 2.1.9 Notation . . . 28
Definition 2.1.10 Konzeptuelles Modell . . . 28
Definition 2.1.11 Sicht . . . 29
Definition 2.2.1 Modelltransformationsspezifikation . . . 39
Definition 2.2.2 Metamodellkonforme Modelltransformationsspezifikation . . . 40
Definition 2.2.3 Abstraktion . . . 40
Definition 2.2.4 Äquivalente konzeptuelle Modelle . . . 40
Definition 2.2.5 Verfeinerung, Verfeinerungsabbildung . . . 40
Definition 2.2.6 Beschreibungstechnik . . . 43
Definition 2.2.7 Komposition . . . 45
Definition 2.2.8 Integrationstransformation . . . 46
Definition 2.2.9 Orthogonale Sichten/Beschreibungstechniken . . . 46
Definition 2.2.10 Verfeinerungskomposition . . . 46
Definition 2.2.11 Teilmodell . . . 47
Definition 2.2.12 Konsistente Erweiterung . . . 48

Definition 3.1.1 Menge der Identifikatoren $\mathbb{ID}$ . . . 51
Definition 3.1.2 Typ . . . 51
Definition 3.1.3 Typ-Belegung (TA) . . . 52
Definition 3.1.4 Klasse (c) . . . 52
Definition 3.1.5 $types(class)$ . . . 53
Definition 3.1.6 Klassenbelegung (CB) . . . 54
Definition 3.1.7 Klassenassoziation (AE) . . . 54
Definition 3.1.8 $oppositeEnd(AE, ae)$ . . . 56
Definition 3.1.9 Objektorientiertes Metamodell (mm) . . . 56
Definition 3.1.10 Isomorphe Metamodelle . . . 57
Definition 3.1.11 Objekt (o) . . . 58
Definition 3.1.12 Objektbelegung (OB) . . . 59
Definition 3.1.13 Objektassoziation (oa) . . . 59
Definition 3.1.14 Objektorientiertes Modell (m) . . . 61

Definition 3.1.15	Menge konformer Modelle ($\mathbb{M}_{mm}$)	62
Definition 3.2.1	Instanzierbares Metamodell	63
Definition 3.2.2	Konfigurationen einer Klasse	64
Definition 3.2.3	Instanzierbarkeits-Gleichungssystem $IES(mm)$	65
Definition 3.2.4	Instanzierbarkeit von Klassen ($instancable(CLASSES, mm)$)	66
Definition 3.2.5	Instanzierbarkeit einer Konfiguration	66
Definition 3.3.1	Variablen $\mathbb{VAR}$	71
Definition 3.3.2	Terme $Term_{TA}$	71
Definition 3.3.3	Identifikatorterme $Term_{\mathbb{ID}}$	71
Definition 3.3.4	Objektvariable *ov*	71
Definition 3.3.5	Objektvariablenbelegung *OVB*	73
Definition 3.3.6	Objektvariablenassoziation *ova*	73
Definition 3.3.7	$getova_{mv}(ovae_0, ovae_1)$	74
Definition 3.3.8	Modellvariable *mv*	74
Definition 3.3.9	Modelltransformationsregel *r*	75
Definition 3.3.10	Modelltransformationsregelwerk *R*	75
Definition 3.3.11	Quell-/Zielmetamodelle, Menge aller möglichen Quell-/Zielmodelle	76
Definition 3.3.12	Parametrisierbare Regeln	77
Definition 3.3.13	Umkehrregel (r^{-1})	78
Definition 3.3.14	Umkehrregelwerk (R_{mm}^{-1})	78
Definition 3.5.1	Modellfragment *mf*	83
Definition 3.5.2	Modellfragment-Match *mfm*	83
Definition 3.5.3	Modellfragment-Match-Menge $MFM(mf, mv)$	84
Definition 3.5.4	Modellfragmentrelation $mfr(mfm, r, mf)$	86
Definition 3.5.5	Modellfragmenttransformation $mft(mfm_0, r)$	89
Definition 3.5.6	Constraint-behaftete Regeln ($constrained(r)$)	90
Definition 3.5.7	Ähnliche Identifikatorterme $oiv_0 \sim oiv_1$, $isTuple(oiv)$	91
Definition 3.5.8	Ähnliche Objektvariablen $ov_0 \sim ov_1$	91
Definition 3.5.9	$attMergeable((a_0, v_0), (a_1, v_1))$	92
Definition 3.5.10	$mergeable(OB_0, OB_1)$	92
Definition 3.5.11	$OBmerge(OB_0, OB_1)$	94
Definition 3.5.12	Merge $mf_0 \cup_{\mathfrak{m}} mf_1$	95
Definition 3.5.13	Teilmodellfragment $mf_0 \trianglelefteq mf_1$	97
Definition 3.5.14	Regelanwendung ($apply(m, r_i)$)	98
Definition 3.5.15	$srcModel(M, r_i)$	98
Definition 3.5.16	Regelwerksanwendung ($transform(\{m_0, \ldots, m_n\}, R)$)	99
Definition 3.5.17	$createsObj(o, ov, r_i, m)$, $createsAsso(asso, ova, r_i, m)$	101
Definition 3.6.1	C1-Modellfragmente	102
Definition 3.6.2	C1-Regeln	103
Definition 3.6.3	C1-Identitäts-Regelwerk $R_{mm_0}^{id}$	103
Definition 3.6.4	Rewrite-Modelltransformationsregelwerk *RRW*	105
Definition 3.6.5	Rewrite-Regelwerksanwendung ($rwTransform(M, RW, (t_0, \ldots, t_n))$)	106
Definition 4.1.1	Anwendbarkeit von Regelwerken	110
Definition 4.1.2	Anwendbarkeit einer Regel	110
Definition 4.1.3	$createsValidFragments(r)$	111
Definition 4.1.4	Ungleichungssystem für gültige Quellfragmente $UGL_{validSrc}$	112

Definition 4.1.5 Deterministische Objektvariable $deterministic(r, ov)$ . . . 115
Definition 4.1.6 $dependsOn(ov, r)$ . . . 116
Definition 4.1.7 $attDependsOn(ov, att, r)$ . . . 117
Definition 4.1.8 *determines*: $(OV^1 \overset{mv_0}{\rightsquigarrow} OV^2)$ . . . 117
Definition 4.1.9 Konfliktfreie Objektvariable $conflictFree(R, ov_i, ov_j)$ . . . 120
Definition 4.1.10 Konfliktgefährdet $potConflicting(R, ov_i, ov_j)$ . . . 121
Definition 4.1.11 $GL'(r, ov_i, mfm_k^0)$ . . . 123
Definition 4.1.12 $PCONF(R, ov_i, ov_j, mfm_k^0, mfm_l^0)$ . . . 124
Definition 4.1.13 Attribut-Gleichungssystem ($EQATTS(R, ov_0, ov_1, mfm_k^0, mfm_l^0)$) . . . 125
Definition 4.1.14 $VALIDOB(R, mm_0, mfm_k^0, mfm_l^0)$ . . . 127
Definition 4.2.1 Metamodellkonformes Regelwerk . . . 131
Definition 4.2.2 $ubConform(mf, mm)$ . . . 132
Definition 4.2.3 $lbConform(mf, mm)$ . . . 132
Definition 4.2.4 $MFM^1(m, r)$ . . . 133
Definition 4.2.5 $numAssos(r, m_0, o, ova, ae_1)$ . . . 133
Definition 4.2.6 Obergrenzenkonformes Regelwerk ($ubConform(R)$) . . . 133
Definition 4.2.7 $maxRedundant(R, r, ova)$ . . . 135
Definition 4.2.8 $maxRelevant(R, r_i, ova)$ . . . 135
Definition 4.2.9 Substrukturen und isomorphe Modellvariablen ($mv_{sub} \sqsubseteq mv$) . . . 136
Definition 4.2.10 $OVP(r_{super}, r_{sub})$ . . . 137
Definition 4.2.11 $redundant(R, r_i, ova)$ . . . 138
Definition 4.2.12 $relevant(R, r_i, ova)$ . . . 140
Definition 4.2.13 $createSameObj(R, \{ov_0, \ldots, ov_n)\})$ . . . 141
Definition 4.2.14 $maxCreateSameObj(R, \{ov_0, \ldots, ov_n\})$ . . . 142
Definition 4.2.15 $cannotCreateSame(R, \{ov_0, \ldots, ov_n\})$ . . . 142
Definition 4.2.16 $\mathbb{OV}_R^{conf}$ . . . 145
Definition 4.2.17 $\mathbb{OV}_R^{conf*}$ . . . 145
Definition 4.2.18 $maxmatch(mv, \{ov_i, ..., ov_j\}, \{ov_k, ..., ov_l\})$ . . . 147
Definition 4.2.19 $maxCard(AE, ae_1, R)$ . . . 151
Definition 4.2.20 $maxVarCard(ova, ae_1, r)$ . . . 152
Definition 4.2.21 $ubVarCard(ova, ae_1, r)$ . . . 153
Definition 4.2.22 Untergrenzenkonforme Regelwerke($lbConform(R)$) . . . 156
Definition 4.2.23 Untergrenzenkonforme Regeln ($lbConform(r)$) . . . 156
Definition 4.2.24 $varLbConform(mv)$ . . . 158
Definition 4.2.25 $minCard(AE, ae_1, r)$ . . . 160
Definition 4.2.26 $minVarCard(ova, ae_1, r)$ . . . 160
Definition 4.2.27 $minmatch(mv, \{ov_i, \ldots, ov_j\}, \{ov_k, \ldots, ov_l\})$ . . . 161
Definition 4.2.28 $lbVarCard(ova, ae_1, r)$ . . . 162
Definition 4.2.29 $lbCard(AE, ae_1, r)$ . . . 164
Definition 4.3.1 Isomorphe Modellfragmente ($mf_0 \sim mf_1$) . . . 167
Definition 4.3.2 Streng bijektive Regelwerke . . . 168
Definition 4.3.3 Bijektive Regelwerke . . . 168
Definition 4.3.4 Quell-Matches . . . 168
Definition 4.3.5 Streng match-bijektive Regelwerke . . . 168
Definition 4.3.6 Match-bijektive Regelwerke . . . 169
Definition 4.4.1 Metamodellkonformes Rewrite-Regelwerk . . . 171

Definition 5.4.1 Elementares Prozessmuster 217
Definition 5.4.2 Elementare Aktivität 217
Definition 5.4.3 Komplexes Prozessmuster 217
Definition 5.4.4 Komplexe Aktivität 218
Definition 5.4.5 Prozessmodell 218
Definition 5.4.6 $canRealize(p,a)$ 223
Definition 5.4.7 Korrekt realisierte Prozessmuster 225

Sätze

Satz 3.2.1 Instanzierbarkeit von Metamodellen . . . 67
Satz 3.5.1 Kommutativität von $\cup_m$ . . . 97
Satz 3.5.2 Assoziativität von $\cup_m$ . . . 97
Satz 3.5.3 Reihenfolge der Anwendung von $\cup_m$-Operationen . . . 97
Satz 3.5.4 Reihenfolge der der Auswahl von Modellfragment-Matches im Quellmodell . . . 98
Satz 3.5.5 Reihenfolge der Anwendung von Regeln eines Regelwerks . . . 99
Satz 3.5.6 Parallelkomposition von Regelwerken . . . 101
Satz 3.5.7 Transformation von Teilmodellen . . . 101

Satz 4.1.1 Verifikationstechnik für *createsValidFragments* . . . 113
Satz 4.1.2 Einfache Verifikationstechnik für *createsValidFragments* . . . 115
Satz 4.1.3 Verifikationstechnik für $\rightsquigarrow$ bei zwei Objektvariablen . . . 118
Satz 4.1.4 Verifikationstechnik für den Determinismus einer Objektvariablen . . . 119
Satz 4.1.5 Verifikationstechnik für Konfliktfreiheit . . . 127
Satz 4.1.6 Anwendbarkeit einer Regel . . . 130
Satz 4.1.7 Anwendbarkeit von Regelwerken . . . 130
Satz 4.2.1 *maxRelevant* impliziert *relevant* . . . 141
Satz 4.2.2 *maxCard* zum Nachweis der Obergrenzenkonformität . . . 152
Satz 4.2.3 Verifikationstechnik für Obergrenzenkonformität . . . 155
Satz 4.2.4 Einfache Verifikationstechnik für die Untergrenzenkonformität einer Regel . . . 159
Satz 4.2.5 Verifikationstechnik für die Untergrenzenkonformität einer Regel . . . 165
Satz 4.2.6 Verifikationstechnik für die Untergrenzenkonformität eines Regelwerks . . . 165
Satz 4.2.7 Verifikationstechnik zum Nachweis von Metamodellkonformität . . . 166
Satz 4.3.1 Match-bijektive Regelwerke sind bijektiv . . . 170
Satz 4.3.2 Strenge Bijektivität von Regelwerken . . . 170
Satz 4.4.1 C1-Identitätsregelwerke sind metamodellkonform . . . 172
Satz 4.4.2 Metamodellkonforme Rewrite-Regelwerke . . . 172

Satz 5.4.1 Prozessmuster realisiert Aktivität . . . 223

Lemmas

Lemma 3.2.1 Instanzierbarkeit einer Konfiguration . . . 67
Lemma 3.2.2 Instanzierbarkeit von Klassen . . . 67
Lemma 3.5.1 Constraint-behaftete Regeln . . . 90
Lemma 3.5.2 Objekte aus ähnlichen Objektvariablen . . . 92
Lemma 3.5.3 Kommutativität von *mergeable* . . . 93
Lemma 3.5.4 Eigenschaften von *mergeable* . . . 93
Lemma 3.5.5 Kommutativität von *OBmerge* . . . 94
Lemma 3.5.6 *OBmerge* von Obermengen von Objektbelegungen . . . 95
Lemma 3.5.7 *OBmerge* erhält Objektidentifikatoren . . . 95
Lemma 3.5.8 Eigenschaften von *mergeable* . . . 95
Lemma 3.5.9 Assoziativität von *OBmerge* . . . 95
Lemma 3.5.10 Merge-Lemma . . . 97
Lemma 3.6.1 C1-Regeln erzeugen C1-Modellfragmente . . . 103

Lemma 4.1.1 $\perp$ als Ergebnis einer Modellfragmenttransformation . . . 112
Lemma 4.1.2 $\rightsquigarrow$ ist reflexiv . . . 117
Lemma 4.1.3 $\rightsquigarrow$ ist transitiv . . . 117
Lemma 4.1.4 Konfliktfreiheit von Objektvariablen . . . 121
Lemma 4.1.5 Einfache Verifikationstechnik für $\neg potConflicting$ . . . 122
Lemma 4.1.6 Einfache Verifikationstechnik für Konfliktfreiheit . . . 122
Lemma 4.1.7 Nachweis von $\neg potConflicting$ . . . 124
Lemma 4.1.8 Konfliktfreiheit für beliebige Quellstrukturen . . . 125
Lemma 4.1.9 Objekte in Objektbelegungen . . . 126
Lemma 4.1.10 Gematchte Quellobjekte . . . 127
Lemma 4.2.1 Nicht metamodellkonforme Zielmodellfragmente . . . 131
Lemma 4.2.2 Matches von Substrukturen in Quellmodellen . . . 136
Lemma 4.2.3 *redundant* impliziert *maxRedundant* . . . 139
Lemma 4.2.4 *cannotCreateSame* impliziert $\neg createSameObj$ . . . 145
Lemma 4.2.5 $\mathbb{OV}_R^{conf*}$ ist Abschätzung für $\mathbb{OV}_R^{conf}$ . . . 146
Lemma 4.2.6 *maxCard* und *maxVarCard* . . . 152
Lemma 4.2.7 *ubVarCard* ist Abschätzung für *maxVarCard* . . . 155
Lemma 4.2.8 Durch eine Modellfragmenttransformation erzeugte Objektassoziationen . . . 156
Lemma 4.2.9 Durch eine Regelanwendung erzeugte Objektassoziationen . . . 157
Lemma 4.2.10 *minCard* ist untere Grenze für die Zahl ausgehender Objektassoziationen . . . 160
Lemma 4.2.11 Abschätzung von *minCard* mit *minVarCard* . . . 161
Lemma 4.2.12 Abschätzung von *minVarCard* mit *lbVarCard* . . . 164
Lemma 4.2.13 *lbCard* ist Abschätzung für *minCard* . . . 164
Lemma 4.2.14 Regelwerke sind kompositional bezüglich ihrer Untergrenzenkonformität . . . 166

Lemma 4.3.1 Streng match-bijektive Regelwerke sind streng bijektiv 169
Lemma 4.4.1 Zielobjektvariablen in C1-Identitätsregelwerken sind konfliktfrei 171
Lemma 4.4.2 Regeln in C1-Identitätsregelwerken sind anwendbar 171
Lemma 4.4.3 C1-Identitätsregelwerke sind anwendbar 171
Lemma 4.4.4 C1-Identitätsregelwerke sind obergrenzenkonform 171
Lemma 4.4.5 Regeln aus C1-Identitätsregelwerken sind untergrenzenkonform 172
Lemma 4.4.6 C1-Identitätsregelwerke sind untergrenzenkonform 172
Lemma 4.4.7 Anwendbarkeit von Rewrite-Regelwerken 172

1 Einführung

1.1 Motivation und Überblick

Die Entwicklung von Software-Systemen ist mittlerweile eine Ingenieursdisziplin, für die sowohl ein wohldefiniertes, strukturiertes Vorgehen, als auch eine angemessene Modellbildung der zu entwickelnden Software entscheidende Erfolgsfaktoren sind. Im Folgenden wird daher ein grundsätzlicher Überblick über die Rolle von Modellen im Software-Engineering gegeben. Hierbei wird zunächst zwischen Modelle zur Beschreibung von Software-Systemen und Modellen zur Beschreibung des Entwicklungsprozesses unterschieden.

1.1.1 Modellierung von Software-Systemen

Wie in vielen anderen Ingenieurswissenschaften bietet der Einsatz von Modellen innerhalb des Software-Engineering die Möglichkeit, von irrelevanten Details zu abstrahieren. Durch die damit einhergehenden verbesserten Strukturierungsmöglichkeiten und die bessere Übersichtlichkeit der komplexen Zusammenhänge von Software-Systemen wird deren Entwicklung erheblich vereinfacht. Werden Modelle von Software-Systemen mit einer geeigneten Semantik versehen, so lassen sich Konsistenzkriterien für sie ableiten, anhand derer sichergestellt werden kann, dass ein Modell ein realisierbares System repräsentiert. Zusätzlich lassen sich für Modelle Operationen zu deren Manipulation definieren, für die sichergestellt werden kann, dass sie die Konsistenz eines Modells nicht verletzen. Diese Operationen stellen mögliche Entwicklungsschritte auf dem Weg zur vollständigen Spezifikation einer Anwendung dar. Hierbei lässt sich das Auffinden sinnvoller Konsistenzkriterien und Modelloperationen durch eine geeignete Einschränkung von Modellen auf die relevanten Sachverhalte zur Beschreibung von Software-Systemen erheblich vereinfachen.

Der Ansatz zur Entwicklung von Software-Systemen unter Verwendung expliziter Modelle wird innerhalb des Software-Engineering als *modellbasierte Entwicklung* bezeichnet. Explizite Modelle verfügen hierbei über eine fest definierte Syntax, welche durch ein Modell zur Beschreibung aller dieser Modelle, einem sogenannten *Metamodell*, festgelegt wird. Modelle zur Beschreibung eines Software-Systems werden *konzeptuelle Modelle* genannt. Ein konzeptuelles Metamodell legt somit die Konzepte fest, die bei der Modellierung eines Software-Systems Verwendung finden. Da die konzeptuellen Modelle in der Regel nicht für eine direkte Bearbeitung durch den Entwickler geeignet sind, werden *Sichten* auf diese Modelle verwendet. Sichten bilden Abstraktionen eines konzeptuellen Modells, welche sich auf bestimmte Aspekte des Gesamtsystems, wie z.B. auf das Verhalten unter Vernachlässigung von interner Struktur, beschränken.

Die Repräsentation von Sichten für den Entwickler erfolgt durch *Beschreibungstechniken*, wie z.B. Automatendiagramme. Durch Beschreibungstechniken können Sichten eines konzeptuellen Modells in einer für den Entwickler verständlichen Notation dargestellt und komfortabel bearbeitet werden. Die einzelnen so erstellten Beschreibungen werden *Artefakte* des Entwicklungsprozesses genannt. In der Regel wird ein modellbasierter Entwicklungsansatz durch ein oder mehrere Werkzeuge, wie beispielsweise durch heutige CASE-Werkzeuge für das implementierungsnahe Design von Anwendungen, unterstützt. Diese erlauben es, Sichten anhand einer geeigneter Notation darzustellen und

ermöglichen ihre Manipulation durch den Benutzer. Die verschiedenen Sichten werden werkzeugintern in ein gemeinsames konzeptuelles Modell der zu erstellenden Software integriert.

Um konzeptuelle Modelle in der Software-Entwicklung gewinnbringend einsetzen zu können, müssen diese jedoch die für den jeweiligen Anwendungsfall angemessene Form der Abstraktion aufweisen. Modellierungssprachen zur Spezifikation von zu erstellenden Systemen, wie z.B. die Unified Modelling Language (UML) [OMG02c], waren bisher zumeist für den universellen Einsatz in unterschiedlichsten Anwendungsdomänen konzipiert. Lediglich für verschiedene technologische Ausprägungen wie z.B. eingebettete Realzeitsysteme existierten bisher angepasste Varianten [SR98, Lyo98]. In letzter Zeit werden jedoch zunehmend Anstrengungen unternommen, die Entwicklung von Software-Systemen durch speziell auf die Anwendungsdomäne zugeschnittene Beschreibungstechniken und konzeptuelle Modelle zu unterstützen.

Für die Auswahl geeigneter konzeptueller Metamodelle und Beschreibungstechniken zur Repräsentation von Systemen spielen demnach zwei Faktoren eine wesentliche Rolle. Zum einen beeinflusst die Anwendungsdomäne, in der ein System erstellt wird, die Art der verwendeten Modelle. So unterscheiden sich die gängigen Konzepte zur Spezifikation von eingebetteten Systemen, wie z.B. Komponente, Aktor, Sensor, etc., grundlegend von denen für die Erstellung von betrieblichen Informationssystemen, bei denen beispielsweise von Geschäftsprozessen, Geschäftsentitäten, etc. die Rede ist. Dementsprechend wird in letzter Zeit gerade im Umfeld der UML verstärkt daran gearbeitet, für verschiedene Anwendungsdomänen jeweils angepasste Arten von konzeptuellen Modellen und Beschreibungstechniken zu entwickeln. Beispiele für den Bereich betrieblicher Informationssysteme finden sich in [OMG02e], [fEFTF01] und [EP00].

Der zweite Faktor, der Einfluss auf die Auswahl der Art eines konzeptuellen Modells zur Beschreibung eines Software-Systems hat, ist die verwendete technische Infrastruktur auf deren Basis ein System realisiert wird. Beispielsweise basieren CORBA-Anwendungen [OMG00] auf einer anderen technischen Infrastruktur als Enterprise Java Beans (EJB)-Anwendungen [Mic04], was bei ihrer Modellierung zu berücksichtigen ist. Im Umfeld der UML existieren hierür eine Reihe von sogenannten *Profilen*, anhand derer die UML und ihre Beschreibungstechniken an unterschiedliche technische Plattformen, wie z.B. Enterprise Java Beans [Gre01] oder CORBA [OMG02d], angepasst werden. Dies erlaubt es, den Code für eine Anwendung in großen Teilen aus technologiespezifischen UML-Modellen zu generieren.

Objektorientierte Modelle sind aufgrund ihrer Nähe zu gängigen objektorientierten Programmiersprachen besonders gut geeignet, um direkt in eine Werkzeugunterstützung umgesetzt zu werden. Somit lassen sich unnötige Brüche zwischen dem konzeptuellen Metamodell und der Werkzeugunterstützung vermeiden. Für das UML-Werkzeug ArgoUML [Col04a] wird der Code für das interne Datenmodell beispielsweise direkt aus der UML-Spezifikation generiert. Für die Spezifikation objektorientierter Metamodelle bietet sich die Meta Object Facitlity (MOF) [OMG02b] der OMG als Spezifikationssprache an. MOF-Metamodelle in Form von Klassenmodellen können einerseits auf eine mathematisch fundierte Semantik abgebildet werden und lassen sich andererseits weitestgehend nahtlos in Anwendungscode für eine Werkzeugunterstützung überführen, wie dies z.B. bei dem Entwicklungswerkzeug AUTOFOCUS [BEH04] der Fall ist. Dementsprechend bieten sich objektorientierte Modelle in besonderem Maße für die Umsetzung eines modellbasierten Entwicklungsprozesses an, falls sie mit einer klaren Semantik versehen sind.

Gerade in den frühen Phasen der Software-Entwicklung, wie dem Requirements Engineering und dem Architekturentwurf einer Anwendung [Som95], ist es sinnvoll, zunächst von vielen technischen Details zu abstrahieren. Zum Beispiel macht es Sinn, Kommunikationsabläufe zwischen Anwendungen zunächst abstrakt und unabhängig von dem später verwendeten konkreten Nachrichtenformat zu beschreiben. Bei dem in der Software-Entwicklung sehr häufig verwendeten Top-Down-Ansatz wird

zunächst eine abstrakte Beschreibung eines Systems erstellt, die später auf ein technologiespezifisches Modell abgebildet wird. Somit werden Modelle schrittweise von abstrakten Spezifikationen hin zu konkreten, plattformspezifischen Systemspezifikationen (im Idealfall Quell-Code) verfeinert.

Ein solches Vorgehen ist auch Gegenstand der von der OMG vorgeschlagenen Model Driven Architecture (MDA) [ORM01]. Die Vorteile dieses Ansatzes liegen im Wesentlichen in der Wiederverwendbarkeit bestehender konzeptueller Modelle auf verschiedenen Abstraktionsebenen, sowie in der Möglichkeit Verfeinerungsschritte zwischen Modellen weitgehend automatisiert durchführen zu können. Die Wiederverwendung abstrakter Systembeschreibungen ist in der Regel sinnvoll, da das technologische Umfeld einer Anwendung zumeist einem deutlich schnelleren Wandel unterzogen ist als das der Anwendungsdomäne. Somit kann bei der Migration einer Anwendung auf die bestehenden Ergebnisse der Analyse der Anwendungsdomäne zurückgegriffen werden. Eine zu weiten Teilen automatisierte Verfeinerung konzeptueller Modelle, bis hin zur Code-Generierung, erspart zum einen manuellen Aufwand, zum anderen vermindert sie, eine korrekte Abbildungsvorschrift vorausgesetzt, die Zahl der durch Inkonsistenzen zwischen den Modellen oder im verfeinerten Modell entstehenden Fehler erheblich.

1.1.2 Prozessmodelle im Software-Engineering

Neben einer geeigneten Modellbildung ist ein klar definierter Entwicklungsprozess der zweite wesentliche Baustein für die ingenieurmäßige Entwicklung von Software-Systemen. Durch die Vorgabe von präzise spezifizierten Entwicklungsschritten kann sichergestellt werden, dass das Modell eines zu entwickelnden Software-Systems vollständig und widerspruchsfrei bleibt. Die Festlegung auf eine sinnvolle Abfolge dieser Entwicklungsschritte bietet einem Entwickler zudem eine Hilfestellung auf dem Weg zur Erstellung des fertigen Systems.

Idealer Weise ist das Vorgehen in einem modellbasierten Entwicklungsansatz auf die zugrundeliegenden konzeptuellen Modelle zur Beschreibung eines Software-Systems abgestimmt. Die Entwicklungsschritte sind in einer für den Entwickler verständlichen Form dokumentiert und beschreiben die Erstellung und Manipulation von Artefakten im Verlauf der Entwicklung. Für verschiedene Arten von Artefakten werden Notationen vorgegeben, in denen die Struktur der jeweiligen Artefakte festgelegt wird. Notationen von Artefakten sind beispielsweise Tabellenvorlagen oder auch die verschiedenen Diagrammarten der UML.

Für die Modellierung von Entwicklungsprozessen bietet sich wiederum der Einsatz geeigneter Metamodelle an. So wird die Menge der im Verlauf eines Entwicklungsvorhabens zu erstellenden Arten von Artefakten und ihre möglichen Beziehungen untereinander in der Regel durch ein so genanntes *Produktmodell* beschrieben. Ein Produktmodell ist demnach ein Metamodell zur Beschreibung der im Rahmen eines Entwicklungsprozesses verwendeten Artefakttypen und deren Beziehungen. Durch die organisationsweite Verwendung eines einheitlichen Produktmodells wird sichergestellt, dass Software-Systeme in einer gleichartigen Weise dokumentiert sind. Dies sichert zum einen die Vollständigkeit einer Entwicklungsdokumentation und erleichtert zum anderen die Einarbeitung in eine bestehende Dokumentation eines Software-Systems, da die verwendeten Artefakttypen und ihre jeweiligen Notationen einheitlich verwendet werden und allen Beteiligten bekannt sind. So definiert der Rational Unified Process (RUP) [Kru00a] beispielsweise für jede Phase der Entwicklung (im RUP als „Workflow“ bezeichnet) eine Menge von zu erstellenden Arten von Artefakten und legt jeweils verantwortliche Rollen für jeden Artefakttyp fest.

Für die Dokumentation des Vorgehens zur Erstellung der Artefakte eignen sich *Prozessmodelle*, in denen neben den einzelnen Artefakttypen die Schritte zu deren Entwicklung und Manipulation sowie deren mögliche Abfolge festgelegt wird. Viele der heute üblichen Ansätze zur Beschreibung von En-

wicklungsaktivitäten und ihrer Zusammenhänge sind eher informell geprägt und weisen lediglich eine semiformale Strukturierung auf. Die Verwendung eines expliziten Prozessmodells verhindert hingegen Mehrdeutigkeiten bei der Abfolge der einzelnen Aktivitäten und erlaubt es zudem, die Korrektheit und Durchführbarkeit eines Entwicklungsprozesses formal zu verifizieren.

1.2 Problemstellung

Innerhalb dieses Abschnitts wird der Umstand erörtert, dass geeignete Techniken zur Transformation von Modellen eine wesentliche Voraussetzung für die Realisierung eines modellbasierten Entwicklungsansatzes ist. Durch die weite Verbreitung objektorientierter Metamodelle zur Spezifikation von konzeptuellen Modellen für die Software-Entwicklung und ihrer sehr guten Eignung als Grundlage für eine Werkzeugunterstützung, kommt dieser Art von Modellen hierbei eine besondere Bedeutung zu.

1.2.1 Modellierung von Software-Systemen

Wie im vorangegangen Abschnitt erörtert wurde, stellt eine geeignete Modellbildung eine wesentliche Voraussetzung für einen effektiven modellbasierten Entwicklungsansatz dar. Geeignete Abstraktionen von konzeptuellen Metamodellen für Software-Systeme müssen sowohl den Besonderheiten der jeweiligen Anwendungsdomäne, als auch denen des technischen Umfeldes, in dem ein System entwickelt wird, Rechnung tragen. Zudem müssen die Metamodelle und die für die Repräsentation ihrer Instanzen verwendeten Beschreibungstechniken den generellen Abstraktionsgrad der Entwicklungsphase, für deren Einsatz sie geschaffen wurden, wiederspiegeln. Dementsprechend gilt es, konzeptuelle Metamodelle zu schaffen, die speziell auf die Erfordernisse der Beschreibung bestimmter Arten von Software-Systeme ausgerichtet sind.

Angesichts der Vielzahl von unterschiedlichen Anwendungsdomänen und technologischen Plattformen zur Realisierung von Anwendungen ist der Einsatz eines universellen konzeptuellen Metamodells, wie beispielsweise das der UML, für die Beschreibung von Software-Systemen kaum realisierbar. Stattdessen bietet es sich an, für die Erstellung abstrakter domänenspezifischer Modelle und feingranularerer technologispezischer Modelle von Software-Systemen jeweils angepasste konzeptuelle Metamodelle zu verwenden.

Aufgrund ihrer leichten Abbildbarkeit auf eine Werkzeugunterstützung sind objektorienterte Modelle und Metamodelle besonders gut für die Modellierung von Software-Systemen geeignet. Für mathematische Kalküle existieren in der Regel Abbildungsvorschriften, die eine konsistente Verfeinerung gewährleisten. Sollen nun objektorientierte, konzeptuelle Modelle für die Spezifikation von Software-Systemen eingesetzt werden, so muss es möglich sein Abbildungen zwischen diesen zu definieren, die Abstraktions- bzw. Verfeinerungsabbildungen der semantischen Interpretation dieser Modelle entsprechen. Hierzu muss für jedes konzeptuelle Metamodell festgelegt werden können, wann zwei seiner Instanzen isomorph sind oder in einer Teilmodellrelation stehen.

Eine automatisierte Verfeinerung von konzeptuellen Modellen findet bislang nur in wenigen Fällen statt. Die hierzu notwendigen Spezifikationen der Abbildungen zwischen den Modellen sind zumeist textuell gefasst und dementsprechend oftmals nicht hinreichend präzise. Innerhalb von Werkzeugunterstützungen für die Software-Entwicklung sind diese Abbildungen in der Regel durch manuell erstellten Code oder aber unter Verwendung von Sprachen zur Transformation einer textuellen Repräsentation von Modellen, wie XSLT [XSL99], realisiert.

Allen diesen Ansätzen ist gemein, dass es nicht möglich ist, formal zu verifizieren, ob eine Transformation zur Laufzeit fehlerfrei ausführbar ist und eine syntaktisch korrekte Instanz des jeweiligen konzeptuellen Zielmetamodells erzeugt. Neben der schlechten Lesbarkeit textueller Spezifikationen und dem hohen Aufwand bei ihrer Erstellung sind die bestehenden Ansätze fehlerträchtig und oftmals nur schwer zu erweitern.

Durch die Verwendung von jeweils an die Anwendungsdomäne und die eingesetzte Technologie angepassten konzeptuellen Metamodellen erhöht sich deren Zahl deutlich. Für die Modellierung von Sichten auf ihre Instanzen durch Beschreibungstechniken bietet es sich daher an, bestehende Notationen wiederzuverwenden. Bei universellen Ansätzen wie der UML findet sich die abstrakte Syntax der verwendeten Notationen direkt im konzeptuellen Metamodell wieder. Die mittels verschiedener Notationen modellierten Informationen sollen stattdessen flexibel in ein konzeptuelles Modell integriert werden können. Dies ist möglich, indem sich die Modelle ihrer abstrakten Syntax auf ein gemeinsames konzeptuelles Modell abgebildet werden. Ähnliche Lösungen werden auch bei der Kopplung von Werkzeugketten für die Software-Entwicklung angestrebt. Eine befriedigende Möglichkeit, solche Modelle zu transformieren und zu integrieren existiert jedoch bis heute nicht.

Demzufolge fehlt für die Spezifikation modellbasierter Entwicklungsprozesse eine Sprache, die es erlaubt, Abbildungen zwischen objektorientierten Modellen präzise und intuitiv verständlich zu definieren. Es gilt eine Möglichkeit zu schaffen, verschiedene Modelle der abstrakten Syntax von Notationen in ein einheitliches konzeptuelles Modell zu integrieren und konzeptuelle Modell automatisiert zu verfeinern. Bestehende Ansätze zur Transformation und Integration objektorientierter Modelle erlauben es bisher jedoch nicht sicherzustellen, dass durch Transformationen immer gültige Instanzen eines gegebenen (konzeptuellen) Zielmetamodells erzeugt werden. Gerade für die automatisierte Ausführung von Transformationen durch eine Werkzeugunterstützung ist die Gewährleistung dieser Eigenschaft jedoch unabdingbar.

1.2.2 Modellierung von Entwicklungsprozessen

Um auch den Entwicklungsprozess für Software-Systeme klar und eindeutig definieren zu können, ist eine geeignete Metasprache zur Beschreibung von Entwicklungsprozessen unerlässlich. Diese muss es ermöglichen, zum einen das Produktmodell einer Organisation zu spezifizieren und zum anderen die zur Erzeugung der Artefakte notwendigen Aktivitäten und deren Verknüpfungen und Abhängigkeiten untereinander festzulegen.

Für einen modellbasierten Entwicklungsansatz ist es zudem nötig, durch die Spezifikation des Entwicklungsprozesses die Auswirkungen auf das Produktmodell und somit indirekt auf das oder die Modelle der zu entwickelnden Software festzuhalten. Während durch gängige Prozessmodelle lediglich die mögliche Abfolge von Entwicklungsschritten spezifiziert werden kann, spielt in einem modellbasierten Entwicklungsprozess die Frage, wie die Ausführung einer Aktivität sich auf ein gegebenes Produktmodell auswirkt, eine zentrale Rolle.

Hierzu lassen sich Entwicklungsaktivitäten als Transformationen einer Instanz des Produktmodells auffassen. Dieser Ansatz erlaubt es, einzelne Aktivitäten als Operationen auf diesem Modell zu definieren. Ein Mechanismus hierfür, der es erlaubt Aussagen darüber zu machen, ob ein so definierter Prozess zu einem für den Prozessentwickler gewünschten Ergebnis führt, existiert bislang jedoch ebenfalls nicht.

1.2.3 Zusammenfassung der Problemstellung

Zusammenfassend lässt sich festhalten, dass für eine erfolgreiche Umsetzung modellbasierter Entwicklungsprozesse jeweils domänen- und technologiespezifische Ausprägungen von konzeptuellen Modellen für die Repräsentation von Software-Systemen auf unterschiedlichen Abstraktionsebenen benötigt werden. Für eine möglichst nahtlose Abbildung in Werkzeuge zur Unterstützung eines solchen Prozesses eignen sich in erster Linie objektorientierte Modelle und Metamodelle. Für die Transformation und Integration dieser Modelle müssen noch geeignete Mechanismen geschaffen werden. Um auch hier keinen konzeptuellen Bruch entstehen zu lassen, muss sich ein solcher Mechanismus ebenfalls an den Konzepten objektorientierter Modelle orientieren. Zudem muss ein Transformationsmechanismus semantisch hinreichend formal fundiert sein, um es zu erlauben, die Ausführbarkeit von Transformationen und die Korrektheit ihrer Ergebnisse für beliebige Eingaben formal nachzuweisen.

1.3 Verwandte Arbeiten

Das Umfeld der modellbasierten Entwicklung auf Basis objektorientierter Modelle ist vor allem geprägt durch die Standardisierungsbemühungen der Object Management Group (OMG) [OMG04]. Im Weiteren werden zunächst einige für diese Arbeit relevante Standards und Initiativen der OMG vorgestellt. Im akademischen Umfeld sind Graphgrammatiken der bisher etablierteste Ansatz zur Transformation graphenartige Strukturen. Da sich auch objektorientierte Modelle als Graphen interpretieren lassen, wird auch dieser Ansatz hier kurz vorgestellt.

1.3.1 Meta Object Facility (MOF)

Durch Metamodelle lassen sich Klassen von Modellen spezifizieren, indem eine Menge gültiger Strukturen dieser Modelle festgelegt wird (siehe auch Abschnitt 2.1.1). Für Metamodelle wird jedoch ebenfalls eine geeignete Syntax benötigt, durch die wiederum die Struktur von Metamodellen definiert wird. Für die Spezifikation von objektorientierten Metamodellen sieht die Object Management Group (OMG) die Meta Object Facility (MOF) [OMG02b] vor. Da im Rahmen dieser Arbeit in erster Linie objektorientierte Modelle für die Spezifikation von Systemen und Entwicklungsprozessen betrachtet werden, ist die MOF hier von besonderem Interesse.

Die MOF sieht vier Ebenen von Modellen vor, zwischen denen Meta- bzw. Instanzierungsbeziehungen bestehen. Abbildung 1.1 stellt diese vier Ebenen graphisch dar. Die innerhalb der MOF als M3 bis M0 bezeichneten Modellebenen umfassen im Einzelnen:

M3-Ebene: Auf dieser Ebene finden sich *Metametamodelle*. So findet sich hier das MOF-Meta-Metamodell zur Spezifikation von Metamodellen.

M2-Ebene: Diese Ebene enthält *Metamodelle*, die anhand eines Metametamodells der Ebene M3, wie z.B. der MOF, definiert werden. Das bekannteste Beispiel eines durch eine MOF-Instanz definierten Metamodells ist das Metamodell der UML [OMG02c]. Innerhalb der MOF werden die Typen der Elemente, aus denen ein Metamodell besteht, sowie deren mögliche Beziehungen untereinander festgelegt. So legt die MOF beispielsweise fest, dass MOF-Metamodelle aus Instanzen des Typs `Class` oder `Association` bestehen können. Diese Elementtypen werden im UML-Metamodell verwendet, um UML-Elemente, wie z.B. `SimpleState` (ein Element des UML-Metamodells vom Typ `Class`), zu definieren.

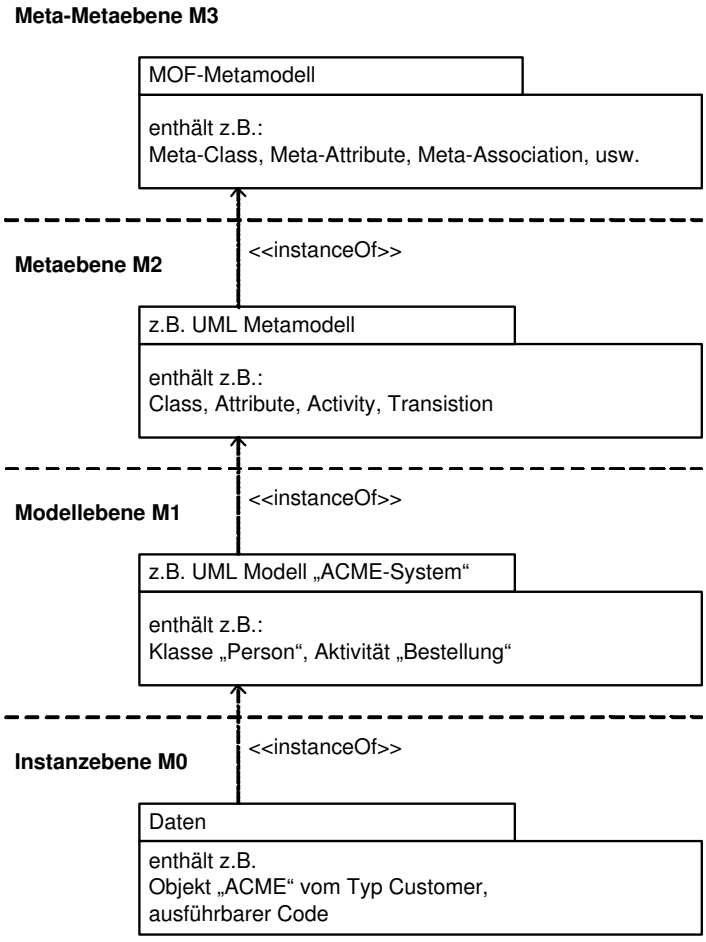


Abbildung 1.1: Die vier Metaebenen der MOF

M1-Ebene: Diese Ebene wird auch als *Modellebene* bezeichnet. M1-Modelle sind Instanzen von Metamodellen der Ebene M2. Hier finden sich beispielsweise UML-Modelle, durch die jeweils ein konkretes Software-System spezifiziert wird. Ein Beispiel für ein Element der M1-Ebene ist ein Objekt `Waiting` als Instanz der Klasse `SimpleState` des UML-Metamodells.

M0-Ebene: Auf dieser untersten Ebene des MOF-Frameworks finden sich die konkreten Daten, die durch Modelle der M1-Ebene beschrieben werden. Ein Beispiel hierfür ist ein konkretes Objekt „ACME Inc.“ als Instanz der Klasse Customer, welches zur Laufzeit innerhalb des Speichers eines realisierten Systems existiert.

Mit der MOF spezifizierte Metamodelle der Ebene M2 sind im wesentlichen Klassenmodelle, für die mit Hilfe der Object Constraint Language OCL, als Teil der MOF, zusätzliche Einschränkungen definiert werden können. Die wesentlichen Elemente von MOF-Metamodellen (Ebene M2) sind dementsprechend Klassen, Assoziationen, Attribute und Datentypen. Auch Vererbung und die Möglichkeit Elemente zu Paketen zusammenzufassen sind von der MOF vorgesehen. Ein bekanntes Beispiel für ein MOF-Metamodell der Ebene M2 ist das UML-Metamodell. Dieses wird innerhalb der UML-Spezifikation anhand eines umfassenden Klassenmodells und einer Reihe von OCL-Constraints definiert.

Die Instanzen von Klassenmodellen sind Objektmodelle. Die Instanzen von Klassen sind Objekte, die über Attribute mit konkreten Werten entsprechend der im Klassenmodell definierten Datentypen verfügen. Ist zwischen zwei Klassen eine Assoziationen definiert, so können Objekte passenden Typs über einen Link vom Typ der Assoziationen miteinander in Beziehung stehen. Zusätzlich zu den in einer Klasse definierten Eigenschaften verfügt ein Objekt über sämtliche Eigenschaften der Klassen, von denen die Klasse des Objekts direkt oder indirekt erbt.

Eine M1-Modell als Instanz eines solchen Metamodells ist demnach ein Objektmodell. Ein konkretes UML-Modell der Ebene M1 zur Spezifikation eines Systems bildet ein Objektmodell als Instanz des UML-Metamodells. Eine weit verbreitete Darstellungsform für UML-Objektmodelle ist der XML Metadata Interchange (XMI) Standard [XMI99] der OMG. Dieser legt fest, wie sich Objektmodelle bei einem gegebenem Metamodell (in diesem Fall das UML-Metamodell) als XML-Dokument [XML00] darstellen lassen. Benutzer der UML werden mit diesem Objektmodell in der Regel jedoch nicht konfrontiert, sie erstellen und bearbeiten UML-Modelle mit Hilfe der verschiedenen UML-Diagrammarten.

Die Instanzen (Ebene M0) eines UML-Modells sind die durch das UML-Modell beschriebenen Elemente eines Software-Systems, wie z.B. Objekte im Speicher oder Prozesse in Ausführung.

Innerhalb der MOF-Spezifikation wird eine Sprache definiert, die es erlaubt M2-Metamodelle in Form von Klassenmodellen und OCL-Constraints zu spezifizieren. Diese Sprache wird anhand eines Meta-Metamodells der Ebene M3 definiert. Die Meta-Metasprache zur Festlegung der MOF ist wiederum die MOF selbst, wobei die Semantik der MOF lediglich in textueller Form festgelegt wird. So wird innerhalb der MOF die Menge der MOF-Klassenmodelle wiederum anhand eines MOF-Klassenmodells spezifiziert. Die Menge aller denkbaren MOF-Metamodelle der Ebene M2 umfasst somit die Menge aller Objektmodelle, welche MOF-Klassenmodelle repräsentieren.

Vor allem die Tatsache, dass M1-Modelle mit einem MOF-konformen Metamodell Objektmodelle darstellen ist für diese Arbeit bedeutsam. Aufgrund dieses Umstands reicht ein einheitlicher Mechanismus zur Transformation von Objektmodellen um beliebige Arten von Modellen, die durch ein MOF-konformes Metamodell spezifiziert wurden, ineinander überzuführen.

1.3.2 Die Unified Modelling Language (UML)

Die Unified Modelling Language (UML) [OMG02c] ist eine graphische Modellierungssprache für den Entwurf von Software-Systemen. Die UML verfügt über neun verschiedene Diagrammarten, deren abstrakte Syntax sich in einem gemeinsamen Modell, dem UML-Metamodell niederschlägt. Im Rahmen dieser Arbeit werden vor allem UML-Klassendiagramme und Objektdiagramme zur Darstellung von Modellen und Metamodellen verwendet.

Da Klassemmodelle auch ein Teil des UML-Metamodells sind, finden sich weite Teile der Strukturen des MOF-Meta-Metamodells innerhalb des UML-Metamodells der Ebene M2 wieder, obwohl das UML-Metamodell tatsächlich eine Instanz des MOF-Modells darstellt. Daher können zur Darstellung dieser M2-Metamodelle ebenfalls UML-Klassendiagramme als „vereinfachte“ Notation verwendet werden, wie dies innerhalb der UML-Spezifikation getan wurde. In [OMG02e] wird ein entsprechendes UML-Profil für die Darstellung von MOF-Metamodellen als Klassendiagramme definiert.

Gerade im Bereich betrieblicher Informationssysteme und objektorientierter Anwendungen hat sich die UML mittlerweile nahezu zu einem de-facto Standard entwickelt. Die weitgehende Akzeptanz der von ihr vorgeschlagenen Notationen ist mit Sicherheit ein wesentlicher Grund hierfür. Ein weiterer bedeutsamer Umstand, der jedoch auch zum Erfolg der UML in der Praxis beigetragen hat, liegt jedoch in der Tatsache begründet, dass das UML-Metamodell in Form eines objektorientierten Klassenmodells vorliegt. Dies erlaubt im Gegensatz zu semiformalen oder mathematischen Modellen eine sehr direkte Umsetzung des UML-Metamodells in objektorientierte Modellierungswerkzeuge.

Aufgrund der weiten Verbreitung der UML als Modellierungssprache für Software-Systeme wird diese von den vielen Ansätzen als Grundlage zur Schaffung domänen-, und technologiespezifischer Modellierungssprachen genommen. Hierzu wird die UML durch Profile um neue Sprachkonstrukte erweitert. Eine solche Erweiterung erfolgt durch die Ableitung neuer UML-Elemente aus bestehenden durch die Einführung von sogenannten „Stereotypen“ und „Tagged Values“.

Allerdings ist eine solche Anpassung der UML mit einer Reihe von Nachteilen verbunden. So ist die Semantik der UML, gerade was Verhaltensaspekte anbetrifft, nur unzureichend definiert [Krü00b]. Eine Erweiterung der UML anhand von Profilen führt in sehr vielen Fällen dazu, dass diese lediglich um neue Diagrammelemente erweitert wird. Die Semantik dieser Elemente schlägt sich in der Regel nicht oder nur unzureichend in der ohnehin nur lückenhaft spezifizierten UML-Semantik nieder. Ein weiterer Nachteil der UML ist die oftmals kritisierte Überfrachtung und Unübersichtlichkeit des UML-Metamodells und seiner Instanzen. Der Grund hierfür liegt darin, dass sich die Elemente der UML-Diagrammtypen zu großen Teilen direkt im UML-Metamodell niederschlagen, da diese zumeist direkt auf das UML-Metamodell abgebildet werden.

Daher ist es in vielen Fällen wünschenswert objektorientierte Metamodelle für die Spezifikation von Software-Systemen unabhängig von der UML zu erstellen. Diese können auf die für einen Anwendungstyp wesentlichen Konzepte beschränkt werden und somit einfacher mit einer formalen Semantik versehen werden.

1.3.3 Model Driven Architecture (MDA)

Die Model Driven Architecture (MDA) [Sol00, ORM01, MM03, KWB03] ist eine Initiative der OMG zur modellbasierten Entwicklung von Software-Systemen. Das Ziel der MDA ist es, die Wiederverwendung von Modellen, wie sie im Verlauf der Entwicklung von Systemen entstehen, zu ermöglichen. Hierzu wird die Verwendung von Modellen zur Spezifikation von Systemen auf verschiedenen Abstraktionsebenen propagiert, die zunächst unabhängig von den eingesetzten Implementierungsplattformen sind. So lässt sich im Idealfall aus einem Modell einer zu erstellenden Anwendung anhand von

einmal definierten Abbildungsvorschriften automatisch eine plattformspezifische Realisierung generieren. Die wesentlichen, durch die MDA formulierten Konzepte für einen modellbasierten Entwicklungsprozess sind im Einzelnen:

Modelle sind eine Repräsentation eines Teils der Funktionalität, Struktur und/oder des Verhaltens eines Software-Systems und seiner Umgebung. Modelle sind „formal“ definiert, was im Kontext der MDA bedeutet, dass die jeweils verwendete Modellierungssprache über eine eindeutig definierte Syntax und Semantik verfügen sollte.

Abstraktion bezeichnet das Weglassen irrelevanter Details in Modellen.

Viewpoint bezeichnet eine Abstraktion eines Modells, die auf einem bestimmten Abstraktionskriterium basiert.

Refinement bezeichnet die Verfeinerung eines Modells durch das Hinzufügen zusätzlicher Information.

Plattform bezeichnet die technologische Infrastruktur auf der eine Anwendung basiert.

Computational Independent Business Model bezeichnet ein Modell eines Systems, in dem von allen Details des Software-Anteils des Systems abstrahiert wird. Es wird auch als *Business Domain Model* bezeichnet.

Platform-Independent Model (PIM) bezeichnet ein Modell der Struktur und Funktion eines Systems, wobei von plattformspezifischen Details abstrahiert wird.

Platform-Specific Model (PSM) bezeichnet ein Model eines Software-Systems unter Berücksichtigung von plattformspezifischen Details.

Innerhalb des MDA-Ansatzes existieren demnach im Wesentlichen drei Abstraktionsebenen für Modelle, die sich am Reference Model for Open Distributed Computing (RM-ODP) orientieren. RM-ODP selbst definiert fünf verschiedene Abstraktionsebenen [ISO95, NW01], jedoch ist auch innerhalb des MDA-Ansatzes die Zahl der möglichen Abstraktionsebenen nicht fest vorgegeben. Beispielsweise können mehrere verschiedene PSMs unterschiedlichen Abstraktionsgrades innerhalb eines MDA-basierten Entwicklungsprozesses verwendet werden. Sowohl für PIMs, als auch PSMs wird von der MDA die UML als Modellierungssprache empfohlen. Es ist jedoch auch der Einsatz anderer Arten von Modellen vorgesehen, insofern deren Syntax eindeutig in Form eines Metamodells definiert ist.

Auf Basis der Metamodelle der Modelle verschiedener Abstraktionsebenen lassen sich Abbildungsvorschriften zwischen den Modellen definieren. So sind Abbildungen zwischen PIMs in der Regel Verfeinerungsschritte, während eine Abbildung eines PIMs auf ein PSM der Abbildung eines plattformunabhängigen Modells auf ein Modell für eine bestimmte Infrastruktur (wie z.B. CORBA) entspricht.

Der Vorteil der Verwendung mehrerer Modelle auf unterschiedlichen Abstraktionsebenen liegt zum einen in der Möglichkeit abstraktere Modelle wiederzuverwenden. Ein plattformunabhängiges Modell einer Anwendung lässt sich auf verschiedene plattformspezifische Modelle abbilden. Dies erlaubt es z.B. bei der Neuentwicklung eines auf einer veralteten Technologie basierenden, Systems auf ein bereits vorhandenes, hinreichend abstraktes Modell des Systems aufzusetzen. Zum anderen erspart die automatisierte Transformation von Modellen zusätzlichen Entwicklungsaufwand und hilft die Konsistenz der Modelle untereinander zu sichern.

Um die UML zur Spezifikation plattformspezifischer Modelle verwenden zu können, muss diese ggf. durch ein Profil um plattformspezifische Konzepte erweitert werden. In dem CORBA-Profil der UML [OMG02d] werden beispielsweise CORBA-Interfaces durch einen speziellen UML-Stereotyp CORBAInterface gekennzeichnet.

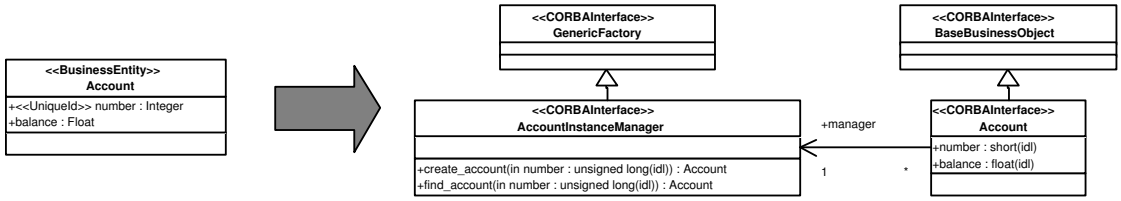


Abbildung 1.2: Exemplarische Abbildung eines plattformunabhängigen Modells auf ein plattformspezifisches Modell

Abbildung 1.2 stellt ein Minimalbeispiel für eine Abbildung eines plattformunabhängigen Modells auf ein CORBA-spezifisches Modell dar. Das Beispiel wurde [ORM01] entnommen. Das Ausgangsmodell ist ein plattformunabhängiges Modell, welches lediglich aus einer Klasse `Account` besteht, die mit dem Stereotyp `BusinessEntity` versehenen ist. Aus dieser Klasse wird ein gleichnamiges CORBA-Interface erzeugt, welches von der Klasse `BaseBusinessObject` erbt. Weiterhin verfügt die neu erzeugte Klasse über einen Verweis auf die Klasse `AccountInstanceManager`, welche von der Klasse `GenericInterface` erbt und Methoden zum Suchen und Anlegen von `Account`-Klassen bietet. Die so erzeugte Struktur stellt also eine mögliche technische Realisierung der `Account`-Klasse auf Basis einer CORBA-Infrastruktur dar.

Derzeit werden die Abbildungen zwischen den Modellen in der Regel durch propreitäre Werkzeuge oder anhand von XSLT-Transformationen [XSL99] der XMI-Darstellung von UML-Modellen realisiert. Eine standardisierte Spezifikationstechnik für die Definition von Abbildungen zwischen MDA-Modellen existiert derzeit noch nicht und ist Gegenstand der im folgenden Abschnitt vorgestellten QVT-Initiative der OMG.

1.3.4 MOF 2.0 Query / Views /Transformations (QVT)

Innerhalb der MDA ist es vorgesehen, zwischen Metamodellen verschiedenen Abstraktionsgrades so genannte „Mappings“ zu definieren, durch die festgelegt wird, wie die Instanzen dieser Metamodelle aufeinander abgebildet werden. Geeignete Techniken zur Spezifikation und automatischen Ausführung von Modelltransformationen sind hierfür die wesentliche Voraussetzung, wie beispielsweise in [GLR+02] festgestellt wird. Da von der MDA die UML als Modellierungssprache empfohlen wird, gilt es demnach Abbildungen zwischen UML-Modellen zu spezifizieren. Zur Definition der UML wurde die MOF verwendet. Somit ist die Transformation von Instanzen von MOF-Metamodellen (also solche der MOF-Ebene M1) eine für die Umsetzung MDA-basierter Entwicklungsansätze entscheidende Technik.

Aus diesem Grunde wurde von der OMG, der zur Erstellung dieser Arbeit aktuelle „Request for Proposal für MOF 2.0 Query / Views /Transformations“ (QVT) [Gro02] erstellt. Ziel des Aufrufs ist die Definition einer einheitliche Sprache, mit deren Hilfe sich Modelle ineinander überführen lassen, deren Metamodelle durch die MOF spezifiziert wurden. Vorschläge zur QVT sollen im Einzelnen die folgenden Anforderungen erfüllen:

- Es soll eine *Sprache zur Abfrage (Query) von Modellen* definiert werden, die es erlaubt, zum einen gezielt Modellelemente aus einem Modell zu filtern und zum anderen Modellelemente als

Quelle für eine Transformation auszuwählen.

- Eine *Sprache zur Spezifikation von Modelltransformationen* soll definiert werden. Diese muss es erlauben, zu spezifizieren, wie verschiedene MOF-Metamodelle zusammenhängen. Es muss möglich sein, anhand einer solchen Spezifikation aus einem Quellmetamodell-konformen Modell ein Zielmodell zu erzeugen, welches wiederum konform zu einem gegebenen Zielmetamodell ist.
- Die *abstrakte Syntax* der definierten Sprache soll in Form eines MOF-Metamodells spezifiziert werden.
- Die Transformationsspezifikationssprache muss hinreichend präzise sein, um eine *automatisierte Ausführung* von Transformationen zu erlauben.
- Die Spezifikationssprache muss es erlauben, *Sichten* auf ein Metamodell zu definieren. D.h. es muss möglich sein, auf Basis bestimmter Kriterien Teile aus Modellen herausfiltern können.
- Die Sprache soll *deklarativ* sein.
- Die Abfrage-, Transformations- und Filtermechanismen müssen auf *Instanzen von MOF-Metamodellen* arbeiten. Im Fall der UML sind dies dementsprechend Instanzen des UML-Metamodells

Als optionale Anforderungen sind für QVT zudem vorgesehen:

- Es soll möglich sein, *bijektive* oder *inverse Transformationspezifikationen* zu erstellen, die es erlauben, Transformationen in beide Richtungen durchzuführen.
- Es sollte die Möglichkeit bestehen die Entstehung von Modellelementen *zurückzuverfolgen*. D.h. für ein Element eines Zielmodells sollten die Elemente des Quellmodells identifizierbar sein, aus denen das Element entstanden ist.
- Mechanismen für *generische* Transformationsspezifikationen sind wünschenswert.
- Es soll möglich sein, Teile einer Transformation mit einem *Transaktionsschutz* zu versehen.
- Ggf. sollen *zusätzliche Informationen* neben dem Quellmodell zur Erzeugung des Zielmodells verwendet werden können.
- Die Sprache soll nach Möglichkeit auch *Rewrite-Transformationen* erlauben, bei denen Ziel- und Quellmodell der Transformation identisch ist.

Mittlerweile existieren acht Vorschläge zur Umsetzung von QVT, die fast alle von verschiedenen Firmen und Firmenverbünden stammen. Ein Vergleich und eine erste Bewertung der verschiedenen Vorschläge wird in [GGKH03] vorgestellt. Als Query-Sprachen wurden sowohl deklarative, imperative, als auch hybride Ansätze vorgeschlagen. Allerdings zeigt sich, dass noch zahlreiche Detailfragen des QVT-Aufrufs weitgehend unklar sind. So existieren beispielsweise unterschiedliche Auffassungen was eine View ist: das Ergebnis einer Transformation oder das Ergebnis einer Query.

Wie auch in [GGKH03] bemängelt wird, sind praktisch alle Ansätze nur sehr informell beschrieben. Eine detaillierte Definition der Semantik von Transformationsspezifikationen fehlt fast immer. Die einzelnen Ansätze zu Modelltransformationen sind teilweise deklartiv, teilweise operational oder sie sehen sogar die Verwendung einer Programmiersprache zur Spezifikation von Transformationen

vor. Drei der acht Vorschläge sprechen sich für OCL als Abfragesprache aus. Lediglich ein Teil der Vorschläge erlaubt bidirektionale Abbildungen zwischen Modellen, in einigen Fällen ist lediglich ein Teil der Regeln einer Modelltransformationsspezifikation bidirektional, während wiederum andere lediglich unidirektionale Abbildungen erlauben.

Zudem beschränkt sich ein Teil der Lösungsvorschläge auf die Transformation von UML-Klassendiagrammen. Eine solche Beschränkung erlaubt jedoch keine Unterstützung für einen generellen modellbasierten Ansatz, bei dem auch andere Modelle als die der UML verfeinert werden sollen. Andere Techniken zeichnen sich zwar durch eine große Vielseitigkeit aus, eine Verifikation der Abbildungsvorschriften hinsichtlich ihrer Durchführbarkeit und der syntaktischen Richtigkeit der durch sie erzeugten Modelle ist jedoch nicht möglich. Die Möglichkeit zu einer solchen Verifikation ist jedoch für eine automatisierte Verfeinerung von Modellen unerlässlich, da ohne sie eine durchgängige Werkzeugunterstützung nicht realisierbar ist.

Zusammenfassend lässt sich sagen, dass die Einreichungen des QVT-Aufrufs bisher noch zu keiner angemessenen Lösung geführt haben. So verfügt ein Großteil der Vorschläge über keine graphische Syntax, oftmals wird kein deklarativer Ansatz verfolgt oder die Spezifikationssprachen beschränken sich lediglich auf die Transformation von UML-Klassendiagrammen.

1.3.5 Graphgrammatiken

Chomsky-Grammatiken sind bereits seid langer Zeit ein bekanntes Mittel um Sprachen über Zeichenketten zu definieren [Cho56, Cho59]. Durch eine solche Grammatik wird die Menge der durch sie erzeugbaren Zeichenketten festgelegt. Analog hierzu definieren Graphgrammatiken [Roz97] die Mengen der durch sie erzeugbaren Graphen. Werden Objektmodelle als Graphen interpretiert, so lässt sich mit Hilfe einer Graphgrammatik jeweils die Menge von Objektmodellen spezifizieren, welche durch sie erzeugt werden kann. Im Gegensatz zu klassischen Chomsky-Grammatiken besteht eine Graphgrammatik aus einer Menge von Produktionsregeln, deren linke und rechte Seite jeweils Graphen sind.

Abbildung 1.3 zeigt ein einfaches Beispiel für eine Graphgrammatik. Durch die Regeln [1] bis [5] lässt sich ein Modell der abstrakten Syntax einer einfachen Variante von Zustandsautomaten erstellen. Eine Produktionsregel kann angewendet werden, falls im Graphen ein Muster mit derselben Struktur wie der Graph der linken Regelseite gefunden wird. Dieser Untergraph des ursprünglichen Graphen wird durch den Graphen der rechten Regelseite ersetzt, wobei durch die Identifikatoren in den Knoten festgelegt wird, welche Knoten bestehenden Knoten im Ursprungsgraphen entsprechen und welche neu im Graph erzeugt werden. Die hier verwendete Notation ist an [RS97] angelehnt, Knoten die nicht neu erzeugt werden und bereits vor der Anwendung der Regel existieren sind zur besseren Lesbarkeit grau unterlegt. Informell besagen die einzelnen Regeln folgendes:

[1] erzeugt aus dem Startsymbol λ einen Startzustand des Typs Start.

[2] fügt eine vom Startzustand ausgehende, neue Transition zu einem ebenfalls neu erzeugten Zustand hinzu.

[3] verbindet einen bestehenden Zustand über eine neue Transition mit einem neuen Zustand.

[4] fügt eine Transition zwischen zwei bestehenden Zuständen in das Modell ein.

[5] fügt eine neue Transition von einem Zustand zum Startzustand ein.

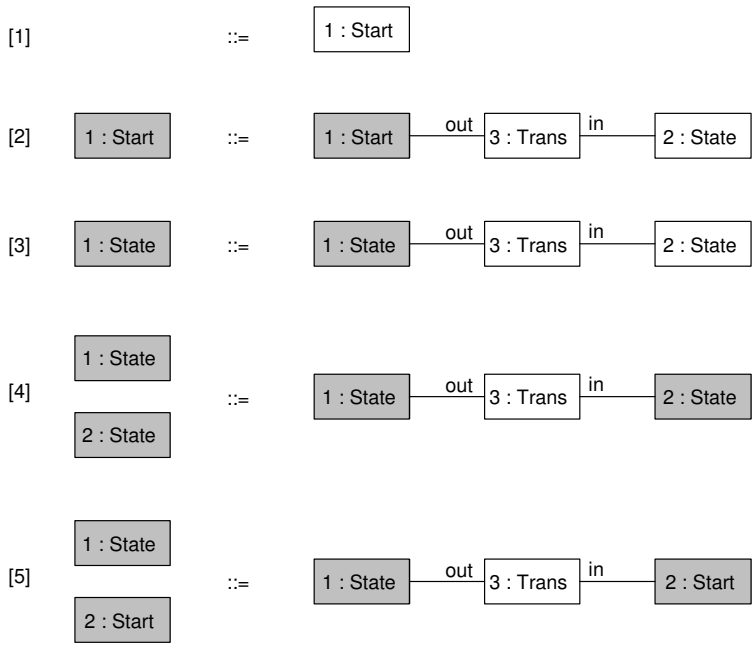


Abbildung 1.3: Beispiel für eine Graphgrammatik

Durch die Produktionsregeln [1] bis [5] ist für die Menge aller durch sie erzeugbaren Automaten bereits sichergestellt, dass lediglich *ein* Startzustand existiert und *jeder* Zustand des Automaten von diesem aus erreichbar ist. Es bleibt jedoch anzumerken, dass alle durch diese Graphgrammatik erzeugbaren Automaten keine Endzustände enthalten. Hierfür wäre die Definition weitere Regeln erforderlich.

Abbildung 1.4 zeigt die Entstehung eines Graph-Modells anhand der Beispielgrammatik. Der Name der Regel, deren Anwendung zu dem jeweiligen Modell führt, ist über dem entsprechendem Modell angegeben. Generell können die Regeln einer Graphgrammatik in beliebiger Reihenfolge ausgeführt werden, vorausgesetzt es findet sich die von einer Regel jeweils geforderte Struktur im Ursprungsgraphen. Ein graphische Repräsentation des fertigen Modells in Form eines Automatendiagramms ist in Abbildung 1.5 dargestellt.

Neben der Spezifikation von Klassen gültiger Graphen können Graphgrammatiken auch zur Transformation von Graphen genutzt werden. Solche *Graph-Transformation*-Ansätze verwenden Graphgrammatiken, anhand derer gültige Quellgraphen gebildet werden können. Zusätzlich zum Aufbau des Quellgraphen wird jedoch parallel ein Zielgraph in geeigneter Weise mit erzeugt. Elemente des Quellgraphen, die auf Elemente es Zielgraphen abgebildet werden, werden hierbei in den Produktionsregeln jeweils über eine Kante verbunden. Somit wird bereits während des Aufbaus des Quellgraphen der entsprechende Zielgraph automatisch mit aufgebaut.

Tripelgraphgrammatiken verwenden einen ähnlichen Ansatz [Sch94]. Jedoch werden hier getrennte Graphgrammatiken zur Definition der beiden Graphklassen verwendet, für deren Regeln jedoch eine paarweise Zuordnung existiert. Der Bezug zwischen beiden Grammatiken wird durch eine dritte Grammatik hergestellt, in welcher der Zusammenhang von Elementen eines Regelpaars festgelegt wird. Somit erlauben es Tripelgraphgrammatiken, Abbildungen zwischen Modellen in beide Richtungen zu interpretieren.

Graphgrammatiken und darauf aufbauende Mechanismen zur Transformation von Graphen basie-

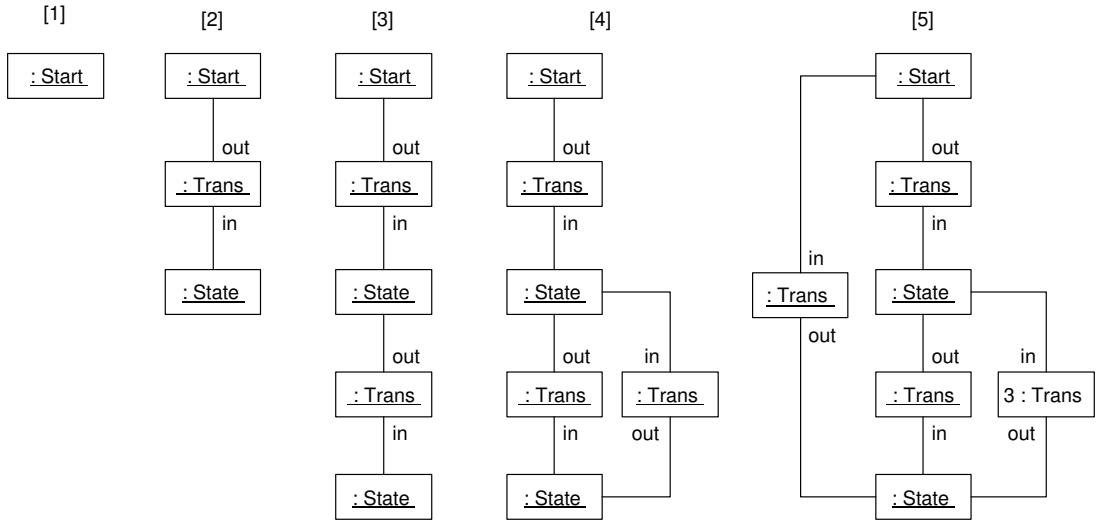


Abbildung 1.4: Entstehung eines Modells durch Anwendung einer Graphgrammatik

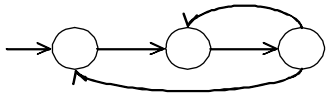


Abbildung 1.5: Graphische Darstellung des in Abbildung 1.4 erzeugten Modells als Automat

ren auf Ergebnissen und Konzepten der Graphtheorie und sind somit hinreichend formal fundiert. Sie werden bereits seit geraumer Zeit in verschiedenen Bereichen des Software-Engineering, wie z.B. die Generierung von graphischen Editoren [Min01] oder die Integration von Software-Werkzeugen [CGN99, NS96] verwendet.

In [BPPT03] wird ein Verfahren skizziert, wie Code-Refactorings durch Graphtransformationen an eine Spezifikation in Form eines UML-Modells propagiert werden können. Verfügt man jedoch über eine geeignete Abstraktionsabbildung zwischen dem Code-Modell und einem UML-Modell, so lässt sich eine abstrakte UML-Spezifikation jederzeit aus einem gegebenen Quellcode generieren.

Dennoch haben sich auf Graphgrammatiken basierende Ansätze zur Transformation von Modellen in der Software-Entwicklung bisher nicht auf breiter Front durchsetzen können.

Ein wesentlicher Grund hierfür ist sicherlich, dass die in Entwicklungswerkzeugen verwendeten Modelle in der Regel objektorientierte Modelle sind. Zwar lassen sich Objektmodelle als attributierte Graphen interpretieren, an einigen Stellen entstehen jedoch Brüche zwischen der Graphentheorie und dem objektorientierten Paradigma.

So werden durch Graphgrammatiken Klassen gültiger Graphen definiert, d.h. eine Graphgrammatik stellt ein Metamodell für eine Menge von Graphen dar. In der Objektorientierung werden jedoch üblicherweise Klassenmodelle als Metamodelle verwendet. Beschränkungen der Kardinalitäten von Assoziationen (bzw. Kanten für Graphen) wie sie in Klassenmodellen in Form von Multiplizitäten vorgesehen sind existieren für die aus Graphgrammatiken erzeugten Graphen per se nicht. Dementsprechend ist es auch nicht möglich die Konformität von durch Graph-Transformationen erzeugten Modellen zu einem objektorientierten Metamodell zu garantieren. Durch sogenannte Graph-Schemata wird versucht, Mengen gültiger Graphen durch eine an Klassenmodelle angelehnte Spezifikationssprache zu definieren. Diese Graphschemata weisen jedoch generell nicht die Mächtigkeit von Klas-

senmodellen auf (siehe auch Abschnitt 6.6.1 zum PROGRES-Ansatz).

Weiterhin sehen die auf Graphgrammatiken basierenden Transformationsansätze keine Transformation primitiver Datentypen, wie sie in Attributen von Objekten zu finden sind, vor. Auch die Berechnung von Objektidentitäten als primitive Typen ist nicht vorgesehen. Daher ist es notwendig den Bezug zwischen Elementen des Quellgraphen und den aus ihnen hervorgehenden Elementen des Zielgraphen entweder durch explizite Kanten, oder aber in Form einer dritten Graphgrammatik im Fall von Tripelgraphgrammatiken, herzustellen. Dies führt wiederum zu einer verschlechterten Lesbarkeit von Transformationsgrammatiken und lässt den Ansatz zunächst wenig intuitiv erscheinen.

Ein Ansatz Graphgrammatiken hin zu einer UML-basierten Spezifikationssprachen für Modelltransformationen zu entwickeln finden sich bereits in [Ake00]. Die genannten, prinzipiellen Schwächen des Einsatzes von Graphgrammatiken zur Transformation bleiben hierbei jedoch erhalten. Ein an Graphgrammatiken angelehnter Ansatz, der diese Mängel vermeidet und sich für die Transformation von Modellen, wie sie der QVT-Aufruf vorsieht, eignet, wird im Rahmen dieser Arbeit vorgestellt

1.4 Ziele und Lösungsansatz der Arbeit

Innerhalb dieses Abschnitts werden die Ziele der Arbeit und der Ansatz zur Lösung der Problemstellung knapp skizziert. Im Anschluss wird der Beitrag der Arbeit für das Feld des Software Engineering diskutiert.

1.4.1 Ziele

Das Ziel der vorliegenden Arbeit ist die Schaffung von Techniken für die Verfeinerung und Integration von Modellen im Software-Engineering, um die Realisierung modellbasierter Entwicklungsansätze zu ermöglichen. Die hier adressierten Ansätze zeichnen sich durch die Verwendung objektorientierter Metamodelle zur Beschreibung von Software-Systemen innerhalb eines eng umrissenen Anwendungsfeldes aus. Die ausschließliche Beschränkung auf die für das Anwendungsumfeld relevanten Konzepte vereinfacht es, diesen Modellen eine explizite Semantik zuzuordnen.

In den verschiedenen Phasen der Software-Entwicklung wird ein System jeweils durch ein konzeptuelles Modell mit einem für die Entwicklungsphase angemessenem Abstraktionsgrad beschrieben. Es soll hierbei möglich sein, automatisierte Verfeinerungsabbildungen zwischen konzeptuellen Modellen anzugeben und die Korrektheit einer Verfeinerung eines Modells zu verifizieren.

Zudem gilt es, verschiedene Beschreibungstechniken flexibel in ein einheitliches konzeptuelles Modell zu integrieren, ohne dass dies Änderungen oder Erweiterungen des konzeptuellen Metamodells nötig macht.

Aufbauend auf den für einen modellbasierten Entwicklungsansatz definierten konzeptuellen Metamodellen und Beschreibungstechniken muss sich ein Entwicklungsprozess definieren lassen. Es soll hierbei möglich sein, das Vorgehen aus bestehenden Vorgehensbausteinen, die für das jeweilige Anwendungsfeld geeignet sind, zu kombinieren. Dementsprechend muss eine Sprache zur Definition von Entwicklungsprozessen hinreichend flexibel sein, um unterschiedliche bestehende Ansätze zu kombinieren. Weiterhin muss die Möglichkeit bestehen, zu verifizieren, ob ein so definierter Prozess ein konsistentes konzeptuelles Modell des zu erstellenden Software-Systems erzeugt.

Da modellbasierte Entwicklungsprozesse in der Regel eine Werkzeuginfrastruktur zu ihrer Unterstützung erfordern, muss gewährleistet sein, dass sich die erarbeiteten Techniken durch entsprechende Werkzeuge automatisieren lassen.

Die Ziele der Arbeit sind somit im Einzelnen:

- Schaffung von Techniken zur Integration von Beschreibungstechniken in Modelle zur Spezifikation von Software-Systemen
- Schaffung von Techniken zur konsistenten Verfeinerung von Modellen von Software-Systemen durch Modelltransformationen
- Schaffung einer Modellierungssprache für flexible Prozessmodelle
- Erbringung des Nachweises, dass die erarbeiteten Techniken sich durch eine geeignete Werkzeugunterstützung umgesetzt werden können

1.4.2 Inhalt

Um einen geeigneten Rahmen für modellbasierte Entwicklungsansätze zu schaffen, werden in dieser Arbeit zunächst grundlegende Begriffe und Konzepte modellbasierter Software-Entwicklung eingeführt. Im einzelnen wird zwischen Prozessmodellen und konzeptuellen Modellen zur Modellierung von Software-Systemen unterschieden. Weiterhin werden Metamodelle als Beschreibungssprache für Modelltypen und semantische Modelle zur Interpretation von Modellen vorgestellt.

Geeignete Techniken zur Transformation von Modellen werden als ein wesentlicher Baustein für die Realisierung eines werkzeugunterstützten, modellbasierten Entwicklungsansatzes identifiziert. Ihr Einsatz im Rahmen eines solchen Entwicklungsansatzes wird daher zunächst allgemein beschrieben, ohne das hierbei bereits eine Festlegung auf objektorientierte Modelle erfolgt.

Es wird ein Prozessmodell zur Spezifikation von Entwicklungsprozessen eingeführt, in dem Entwicklungsschritte in Form von Transformationen einer Instanz des Produktmodells spezifiziert werden. Dies erlaubt es, Vorgehensbausteine flexibel zu kombinieren und Aussagen über die Vollständigkeit eines so erstellten „methodischen Baukastens“ zu machen.

Für konzeptuelle Modelle wird ein Verfeinerungs- und Abstraktionsbegriff erarbeitet. Dies erlaubt es zu verifizieren, ob eine Modelltransformation eine gültige Verfeinerungsabbildung bezüglich einer gegebenen Abstraktionsbeziehung zwischen zwei Arten von Modellen darstellt.

Die Integration von Artefakten des Prozessmodells in ein gemeinsames konzeptuelles Modell erfolgt im hier vorgestellten Ansatz durch eine Abbildung von Instanzen der abstrakten Syntax der verwendeten Notation in das jeweilige konzeptuelle Modell. Es wird formal festgehalten, wann Beschreibungstechniken orthogonale Aspekte eines Systems spezifizieren, so dass die Entstehung von Inkonsistenzen durch die Integration dieser Artefakte ausgeschlossen werden kann. Weiter wird dargestellt, wie sich verifizieren lässt, ob bei der Integration von Artefakten in ein verfeinertes konzeptuelles Modell die ursprüngliche Abstraktionsbeziehung zwischen diesem Modell und dem abstrakten Modell erhalten bleibt. Diese Techniken sind entscheidend, um innerhalb eines Entwicklungsprozesses die Konsistenz von konzeptuellen Modellen unterschiedlichen Abstraktionsgrades sicherzustellen.

Zur Transformation konzeptueller Modelle, der Integration von Beschreibungstechniken und für die Transformationen von Instanzen eines Produktmodells wird eine eigene Modelltransformationssprache vorgestellt, die es erlaubt, objektorientierte Modelle zu transformieren. Die Bidirectional Object-Oriented Transformation Language (BOTL)[1] erlaubt die Spezifikation von Abbildungen zwischen mehreren Quellmodellen auf ein Zielmodell. Die Sprache basiert auf einem mathematischen Modell für objektorientierte Modelle und Metamodelle. Neben einer formalen Definition der Eigenschaft, dass ein Modell konform zu einem gegebenen Metamodell ist, wird auch eine Technik eingeführt, mit

[1] Die in dieser Arbeit vorgestellte Sprache BOTL basiert auf den gemeinsam mit Peter Braun erarbeiteten und in [BM03a] veröffentlichten Ergebnissen.

der sich auch für sehr restriktive Metamodelle nachweisen lässt, dass es endliche Instanzen für dieses Metamodell gibt.

BOTL verfügt über eine formale Semantik für die Ausführung von Transformationen. Transformationen zwischen Modellen werden in BOTL durch Mengen von Regeln spezifiziert. Eine Regel legt jeweils fest, wie Ausschnitte eines Quellmodells auf ein Fragment des Zielmodells abgebildet werden. Für die Angabe dieser Regeln steht eine graphische, an die UML angelehnte Syntax zur Verfügung.

Darauf aufbauend werden Verifikationstechniken vorgestellt, die es zum einen ermöglichen, für BOTL-Spezifikationen zu verifizieren, dass diese *anwendbar* sind, d.h. ihre Ausführung verläuft fehlerfrei und führt zu einem deterministisch Ergebnis. Zum anderen kann verifiziert werden, dass BOTL-Spezifikationen für beliebige Quellmodelle immer ein zu seinem Metamodell konformes Zielmodell erzeugen, sofern auch die Quellmodelle konform zu ihren Metamodellen sind. Diese Eigenschaft wird als *Metamodellkonformität* bezeichnet. Die Techniken zum Nachweis der Metamodellkonformität werden benötigt, um in einem Entwicklungsprozess sicherstellen zu können, dass die (automatisierte) Verfeinerung konzeptueller Modelle in keinem Fall zu inkonsistenten Ergebnissen führt. Bei der Integration von Artefakten in ein konzeptuelles Modell können mögliche Konflikte zwischen einzelnen Artefakten so bereits während der Definition des Entwicklungsprozesses identifiziert werden.

Anhand weiterer Verifikationstechniken lässt sich nachweisen, dass eine BOTL-Spezifikation bezüglich eines Quellmetamodells bijektiv ist. Ist dies der Fall, so erzeugt die Umkehrung der Regeln eines BOTL-Regelwerks aus einem generierten Zielmodell wieder das jeweilige Quellmodell oder ein hierzu isomorphes Modell. Bijektive Transformationen erlauben es zum einen, Artefakte aus konzeptuellen Modellen zu erzeugen, zum anderen erhält man für bijektive Abstraktionsabbildungen automatisch eine gültige Verfeinerungsabbildung in Form der Umkehrabbildung.

Um Entwicklungsschritte in Form von Transformationen eines bestehenden Produktmodells modellieren zu können, wird BOTL um die Möglichkeit erweitert, Rewrite-Transformationen, d.h. Transformationen *eines* bestehenden Modells, zu spezifizieren. Die Korrektheit und Metamodellkonformität dieser speziellen BOTL-Regelwerke lässt sich ebenfalls formal verifizieren.

Im Rahmen dieser Arbeit wird die KOGITO-Methodik für das Requirements Engineering webbasierter B2B Anwendungen vorgestellt. Anhand dieser Methodik wird exemplarisch skizziert, wie die Techniken für eine modellbasierte Entwicklung mit Hilfe der Transformationssprache BOTL umgesetzt werden können.

Die Realisierbarkeit und Funktionsfähigkeit der Techniken zur Transformation von Modellen und zur Verifikation von Transformationsregelwerken werden anhand eines im Rahmen dieser Arbeit entstandenen Werkzeuges belegt. Die vorgestellte Werkzeugunterstützung erlaubt die graphische Spezifikation von BOTL-Regelwerken und -Metamodellen, sowie die automatisierte Verifikation ihrer Metamodellkonformität. Weiterhin verfügt das Werkzeug über einen Testmodus, in dem sich Objektmodelle graphisch spezifizieren und transformieren lassen. Das Ergebnis der Transformation wird ebenfalls in Form eines Objektmodells dargestellt. Das Werkzeug ermöglicht auch eine Transformation von Modellen in technologiespezifischen Formaten. Derzeit lassen sich Java-Objektgeflechte transformieren, eine Erweiterung zur Transformation von XMI-Modellen, welche die Integration von UML-Werkzeugen ermöglicht, ist derzeit in der Entstehung begriffen.

Zusammenfassend werden im Rahmen dieser Arbeit die folgenden Ergebnisse erbracht:

- Erarbeitung grundlegender Konzepte für den Einsatz von Modelltransformationen für die Integration von Beschreibungstechniken in konzeptuelle Modelle, die Verfeinerung dieser Modelle und die Modellierung flexibler Entwicklungsprozesse

- Ein mathematisch fundiertes Modell für objektorientierte Modelle und Metamodelle;

- Ein Verfahren zur Verifikation der Instanzierbarkeit objektorientierter Metamodelle
- Die mit einer formalen Semantik versehene Modelltransformationssprache BOTL
- Formale Verifikationstechniken für den Nachweis der Anwendbarkeit, der Metamodellkonformität und der Bijektivität von BOTL-Spezifikationen
- Ein Mechanismus, der es erlaubt auch Rewrite-Transformationen auf einem bestehenden Modell zu spezifizieren und ihre Anwendbarkeit und Metamodellkonformität zu verifizieren
- Demonstration der erarbeiteten Konzepte für einen modellbasierten Entwicklungsansatz anhand eines Fallbeispiels
- Eine Werkzeugunterstützung für die graphische Spezifikation von BOTL-Spezifikation, die automatisierte Verifikation von BOTL-Spezifikationen, die Erprobung von Transformationen anhand eines graphischen Editors und die Transformation objektorientierter Modelle in unterschiedlichen technischen Formaten anhand einer BOTL-Spezifikation ermöglicht

1.4.3 Beitrag der Arbeit

Die im Rahmen dieser Arbeit erbrachten Ergebnisse erlauben es, spezifisch angepasste, modellbasierte Entwicklungsprozesse für unterschiedliche Arten von Software-Systemen zu definieren. Die Einschränkung einer so spezifizierten Methodik auf ein fest umrissenes Anwendungsfeld und ggf. eine Technologie ermöglicht es, die Entwicklung eines Software-Systems wesentlich gezielter und effizienter zu unterstützen, als dies bei universelleren Ansätzen möglich ist. So spiegeln sich in den verwendeten konzeptuellen Metamodellen genau die für die System-Entwicklung relevanten Konzepte, wie z.B. „Geschäftsprozess“, „Aktor“, etc., wieder. Dies erlaubt es, Systeme in der Sprache der Anwendungsdomäne zu modellieren, wobei sich durch die Auswahl geeigneter domänenspezifischer Metamodelle der Lösungsraum gezielt auf eine Klasse sinnvoller Lösungen beschränken lässt.

Wie auch von der MDA vorgeschlagen werden im Verlauf der Entwicklung konzeptuelle Modelle auf verschiedenen Abstraktionsebenen zur Spezifikation eines Software-Systems verwendet. Die Verfeinerungsabbildungen können durch BOTL-Transformationen automatisiert durchgeführt werden. Somit lässt sich das Wissen um geeignete Strategien zur Verfeinerung dieser Modelle in Transformationsspezifikationen kapseln. Beispielsweise kann bei der Abbildung eines PIM auf ein PSM die Verwendung geeigneter Entwurfsmuster im PSM bereits durch die Verfeinerungsspezifikation sichergestellt werden. Aufrund der graphischen und intuitiven Syntax der BOTL können solche Abbildungen mit Hilfe eines Werkzeuges, wie es in Teil 6 beschrieben wird, einfach und schnell erstellt und auf ihre Korrektheit hin verifiziert werden.

Durch die Möglichkeit, verfeinerte konzeptuelle Modelle nach ihrer Weiterbearbeitung daraufhin zu überprüfen, ob sie immer noch korrekte Verfeinerungen des ursprünglichen Modells darstellen, kann die Konsistenz einer Systemdokumentation gewährleistet werden. Darüber hinaus kann sichergestellt werden, dass ein System als letzte Stufe der Verfeinerungen konsistent zu den, in abstrakteren Modellen formulierten, Anforderungen ist.

Die Transformation der Inhalte von Artefakten in Sichten eines konzeptuellen Modells erlaubt die Entkoppelung verwendeten Notationen von den konzeptuellen Metamodellen. Dies ermöglicht es, flexibel unterschiedliche Notationen innerhalb eines Entwicklungsprozesses zu verwenden und eine Methodik einfach um zusätzliche Beschreibungstechniken zu erweitern. In der Praxis lassen sich so unterschiedliche Modellierungswerkzeuge für die Erstellung von Artefakten verwenden, deren Inhalte in ein gemeinsames konzeptuelles Modell integriert werden.

Durch die Beschränkung auf die innerhalb einer Domäne benötigten Konzepte, sowie das Fehlen notationsspezifischer Elemente in den konzeptuellen Metamodellen sind diese in der Regel deutlich kleiner und übersichtlicher als beispielsweise das Metamodell der UML. Dies vereinfacht die Angabe einer geeigneten formalen Semantik für ein solches Metamodell.

Die Verwendung des im Rahmen dieser Arbeit vorgestellten Prozessmusteransatzes trägt weiter zur Flexibilisierung des Entwicklungsprozesses bei. Er erlaubt es, Vorgehensbausteine auf Basis eines einheitlichen Produktmodells flexibel miteinander zu kombinieren, und ermöglicht es Entwicklungsprojekten somit, umgehend auf ein geändertes Umfeld oder sich ändernde Anforderung zu reagieren.

1.5 Aufbau der Arbeit

In Kapitel 2 werden zunächst die Grundlagen für modellbasierte Entwicklungsprozesse vorgestellt. Abschnitt 2.1 führt die hierzu notwendige Begrifflichkeiten zu Prozessmodellen und konzeptuellen Modellen ein. Innerhalb von Abschnitt 2.2 wird ein Klassifikationschema für Spezifikationssprachen für Modelltransformationen angegeben. Weiterhin wird die Rolle von Modelltransformationen im Rahmen eines Entwicklungsprozesses diskutiert, ohne jedoch explizit von objektorientierten Modellen auszugehen.

Kapitel 3 stellt die „Bidirectional Object-Oriented Transformation Language“ (BOTL) zur Transformation objektorientierter Modelle vor. Hierzu wird in Abschnitt 3 zunächst ein mathematisch fundiertes Modell für Klassen- und Objektmodelle eingeführt. Im Anschluss (Kapitel 3.2) wird ein Verfahren vorgestellt, mit dem sich ggf. auch für sehr restriktive Metamodelle nachweisen lässt, dass diese instanzierbar sind, d.h. dass endliche Modelle existieren, welche konform zu dem jeweiligen Metamodell sind. Aufbauend auf diesem mathematischen Modell werden in Abschnitt 3.3 Regeln und Regelwerke zur Transformation objektorientierter Modelle und in Abschnitt 3.4 ein UML-Profil für ihre Darstellung eingeführt. Die Semantik von BOTL-Regelwerken wird innerhalb von Abschnitt 3.5 formal auf Basis des mathematischen Modells für objektorientierte Modelle spezifiziert. Im Anschluss wird in Abschnitt 3.6 eine Erweiterung der BOTL eingeführt, die es erlaubt auch Rewrite-Transformationen, die ein bestehendes Modell verändern, zu spezifizieren.

Kapitel 4 führt schließlich eine Reihen von Eigenschaften von BOTL-Regelwerken ein. So wird diskutiert, wann ein Regelwerk prinzipiell ausführbar ist (Abschnitt 4.1), wann es mit Sicherheit nur metamodellkonforme Modelle erzeugt (Abschnitt 4.2) und wann ein Regelwerk bijektiv ist (Abschnitt 4.3). Für alle Eigenschaften werden Verifikationstechniken vorgestellt, die es erlauben, diese Eigenschaften auch automatisiert zu verifizieren. Abschnitt 4.5 diskutiert anhand eines Beispiels, wie sich Aussagen über die Semantik der durch ein BOTL-Regelwerk erzeugten Zielmodelle machen lassen. Abschnitt 4.6 ordnet die Transformationssprache BOTL schließlich anhand des in Abschnitt 2.2.1 vorgestellten Klassifikationsschemas ein.

Innerhalb von Kapitel 5 wird die Realisierung eines modellbasierten Entwicklungsprozesses mit Hilfe der BOTL anhand eines Beispiels demonstriert. Hierzu wird in Abschnitt 5.1 zunächst die KOGITO-Methodik für das Requirements Engineering web-basierter Anwendungen vorgestellt. In Abschnitt 5.2 wird demonstriert, wie sich konzeptuelle Modelle konsistent verfeinern lassen. Innerhalb von Abschnitt 5.3 wird gezeigt, wie sich mit Hilfe von BOTL KOGITO-Beschreibungstechniken in ein konzeptuelles KOGITO-Modell integrieren lassen. Abschnitt 5.4 zeigt exemplarisch auf, wie sich mit BOTL Vorgehensschritte als Transformationen einer Instanz eines Produktmodells spezifizieren lassen.

Kapitel 6 stellt schließlich die im Rahmen dieser Arbeit entstandene Werkzeugunterstützung für die Spezifikation, Verifikation und Ausführung von BOTL-Transformationsregelwerken vor. Abschnitt

6.1 fasst zunächst die wesentlichen die Anforderungen an das Werkzeug zusammen und gewährt einen grundsätzlichen Überblick über den Aufbau und die Funktionsweise der Werkzeugunterstützung. Abschnitt 6.2 stellt die Infrastruktur für die Werkzeugunterstützung vor, in der BOTL-Konzepte wie Regelwerk, Modell und Metamodell umgesetzt wurden. Der Editor zur graphischen Spezifikation von Regelwerk ist Gegenstand von Abschnitt 6.3. Abschnitt 6.4 geht auf die Komponente zur Verifikation von Regelwerken ein und Abschnitt 6.5 stellt schließlich die Komponente zur Transformation von Objektmodellen vor. Abschließend wird in Abschnitt 6.6 noch eine Reihe verwandter Werkzeuge und Ansätze vorgestellt und diskutiert.

Kapitel 7 fasst die erzielten Ergebnisse nochmals überblicksartig zusammen (Abschnitt 7.1) und gewährt in Abschnitt 7.2 einen Ausblick auf Erweiterungsmöglichkeiten und weitere mögliche Fragestellungen im Zusammenhang mit den erbrachten Ergebnissen.

2 Grundlagen

2.1 Modellbasierte Software-Entwicklung

Ansätze zur modellbasierten Software-Entwicklung zeichnen sich durch die Verwendung expliziter Modelle für die Beschreibung des Entwicklungsprozesses einerseits und die Beschreibung des zu erstellenden Systems andererseits aus [SPHP02]. In den folgenden Abschnitten werden zunächst die Begriffe Modell, Metamodell und semantisches Modell eingeführt. Anschließend wird eine Unterscheidung zwischen Prozessmodellen und konzeptuellen Modellen getroffen. Während Prozessmodelle dazu dienen, den Entwicklungsprozess zu definieren, werden konzeptuelle Modelle dazu verwendet, das zu erstellende System selbst abzubilden.

2.1.1 Modelle und Metamodelle

Der Einsatz von Modellen zur Beschreibung von Software-Systemen ermöglicht es von unwesentlichen Details zu abstrahieren und sich auf wesentliche Aspekte des zu erstellenden Systems zu konzentrieren. Dies erlaubt es dem Entwickler eines Software-Systems, leichter die Übersicht über die komplexen Strukturen der konkreten Anwendung zu behalten und vereinfacht somit den Entwicklungsprozess.

Gemäß [Sta73] zeichnet sich ein Modell im wesentlichen durch drei Eigenschaften aus:

1. Modelle sind stets Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.
2. Modelle erfassen im allgemeinen nicht alle Eigenschaften des Originals, welches sie repräsentieren, d.h. sie abstrahieren von Eigenschaften des durch sie beschriebenen Originals.
3. Modelle erfüllen eine bestimmte Funktion.

In Anlehnung an diesen Modellbegriff werden Modelle, wie sie für das Software-Engineering benötigt werden, im Rahmen dieser Arbeit folgendermaßen definiert:

Definition 2.1.1 (Modell)
Ein *Modell* beschreibt abstrakt einen existierenden oder zu realisierenden Teil der Realität.
Die Menge aller Modelle wird mit $\overline{\mathbb{M}}$ bezeichnet. ❍

Modelle verfügen in der Regel über eine explizite Syntax, durch welche die Menge der möglichen Modelle eingeschränkt wird. Diese Syntax wird durch ein sogenanntes *Metamodell* festgelegt. Werden Modelle für die Beschreibung von Software-Systemen verwendet, so wird durch ein geeignetes Metamodell der Lösungsraum für die Realisierung eines Software-Systems so eingeschränkt, dass sich zahlreiche Fehler bei der Entwicklung von vornherein ausschließen lassen.

Für verschiedene Arten von Modellen wird durch Metamodelle festgelegt, wie ein korrektes Modell beschaffen sein muss. Je nach Art der Modelle können Metamodelle beispielsweise in Form mathematischer Definitionen, Chomsky-Grammatiken [Cho56] oder Klassenmodellen für objektorientierte

Modelle auftreten. Die nachfolgende Definition führt die Begriffe *Metamodell* und *Metametamodell* ein.

Definition 2.1.2 (Metamodell, Metametamodell)
Ein *Metamodell* ist ein Modell, welches eine Klasse von gültigen Modellen definiert.

Die Menge aller Metamodelle wird mit $\overline{\mathbb{MM}} \subset \overline{\mathbb{M}}$ bezeichnet. $\overline{\mathbb{M}}_{mm}$ bezeichnet die Menge aller durch das Metamodell *mm* definierten Modelle. Ein Modell $m \in \overline{\mathbb{M}}_{mm}$ wird auch als *Instanz* des Metamodells *mm* bezeichnet.

Dementsprechend ist ein *Metametamodell* ein Modell, welches eine Klasse gültiger Metamodelle definiert. Es gilt weiterhin: $\overline{\mathbb{M}}^2_{mm} := \bigcup_{mm' \in \overline{\mathbb{M}}_{mm}} \overline{\mathbb{M}}_{mm'}$. ❍

Entsprechend der Definition sind somit auch Metamodelle Modelle, welche jedoch einem speziellen Zweck, nämlich der Spezifikation einer Menge von Modellen, dienen. Daher gilt die Relation $\overline{\mathbb{MM}} \subset \overline{\mathbb{M}}$. Im Umfeld der UML und der MOF (siehe Abschnitt 1.3.1) äußert sich dieser Sachverhalt beispielsweise darin, dass auch UML-Klassenmodelle prinzipiell MOF-Objektmodelle darstellen, für die jedoch eine angepasste Notation, nämlich UML-Klassendiagramme, existiert.

Die Menge $\overline{\mathbb{M}}^2_{mm}$ enthält alle Modelle, die Instanzen der durch das Metametamodell *mm* definierten Metamodelle sind. Ein regulärer Ausdruck (siehe beispielsweise [Sch93]) kann als ein Metamodell interpretiert werden, da durch ihn eine Menge von Zeichenketten spezifiziert wird. Wird durch das Metametamodell *ra* die Menge aller regulären Ausdrücke spezifiziert, so enthält die Menge $\overline{\mathbb{M}}^2_{ra}$ alle denkbaren Zeichenketten, welche durch reguläre Ausdrücke beschrieben werden können.

Dementsprechend gilt im Umfeld der Objektorientierung für ein Metametamodell *cm*, welches die Menge aller Klassenmodelle definiert, beispielsweise:

$\overline{\mathbb{M}}^2_{cm} =$ „Die Menge aller denkbaren Objektmodelle“

Für die Entwicklung von Software-Systemen existiert eine Vielzahl unterschiedlicher Arten von Modellen. Während für *semiformale Modelle* lediglich formal festgelegt ist, wann ein Modell syntaktisch korrekt ist, verfügen *formale Modelle* über eine explizite Semantik. Eine Semantik ordnet einem Modell eine Bedeutung in Form von Aussagen über den durch das Modell beschriebenen Sachverhalt zu. Idealerweise liegt die Semantik eines Modells in Form einer allgemein anerkannten, eindeutigen Notation vor. Die UML kann demnach als semiformale Modellierungssprache verstanden werden, da zwar ihre Syntax hinreichend formal spezifiziert ist, die Semantik der verwendeten Modellkonstrukte jedoch oftmals nur sehr unpräzise in textueller Form festgelegt ist [EK99, FELR98].

Definition 2.1.3 (Semantik, semantisches Modell)
Die Semantik für eine Art von Modellen legt fest, wie ein Modell auf ein semantisches Modell abgebildet wird. Ein semantisches Modell ist ein Modell, dessen Interpretation als Beschreibung der Realität zweifelsfrei möglich ist. ❍

Die Semantik eines Modells legt also fest, wie ein Modelle als Abstraktion der Realität zu interpretieren ist. Mögliche Arten von semantischen Modellen werden jeweils in einer Sprache ausgedrückt, deren Interpretation zweifelsfrei ist. Das Spektrum reicht hierbei von mathematischen fundierten Modellarten, die mit Formalismen wie FOCUS [BS01] oder CSP [Hoa85] beschrieben werden, bis hin zu Quellcode in einer existierenden Programmiersprache, wie z.B. Java.

Abbildung 2.1 skizziert den Zusammenhang zwischen Modellen, Metamodellen und semantischen Modellen. Durch ein Metamodell wird eine Klasse gültiger Modelle definiert. Eine Semantik auf Basis

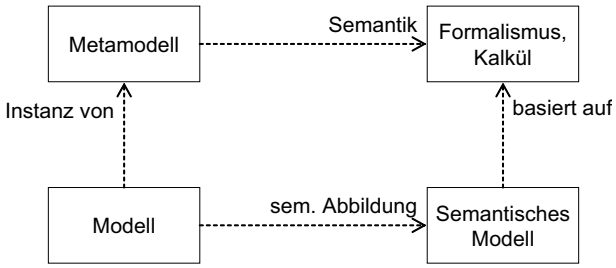


Abbildung 2.1: Zusammenhang zwischen Modell, Metamodell und semantischem Modell

des Metamodells gibt an, wie die Elemente einer Instanz des Metamodells im Sinne eines Formalismus oder Kalküls zu interpretieren sind.

Anhand der Abbildung eines Modells auf ein semantisches Modell ist es möglich einem Modell eine Bedeutung zuzuordnen. Zudem kann zu einem Metamodell eine Menge von möglichen Operationen zur Manipulation seiner Instanzen angegeben werden, von denen bekannt ist, dass ihre Interpretation anhand der Semantik wieder zu sinnvollen Ergebnissen führt. Somit können für Entwickler oder eine Werkzeugunterstützung für die Software-Entwicklung Modelloperationen vordefiniert werden, die sinnvollen Entwicklungsschritten entsprechen.

2.1.2 Prozessmodelle

Prozessmodelle beschreiben den Entwicklungsprozess von Systemen. Die wesentlichen Elemente von Prozessmodellen sind Entwicklungsartefakte und Aktivitäten zur Erzeugung oder Manipulation dieser Artefakte.

Entwicklungsartefakte sind Dokumente oder Teile von Dokumenten, welche im Verlauf der Software-Entwicklung erstellt und erweitert werden. Ein Artefakt ist hierbei in einer vordefinierten Notation verfasst, die durch ihren jeweiligen Dokumenttyp festgelegt ist. Notationen werden innerhalb von Abschnitt 2.1.3 vorgestellt.

Definition 2.1.4 (Artefakt)
Ein *Artefakt* besteht aus

- einem eindeutigen Bezeichner und
- einem Artefakttyp. ❍

Während Artefakte generell auch informelle, textuelle Beschreibungen oder Skizzen sein können, sind im Rahmen einer modellbasierten Software-Entwicklung lediglich Artefakte von Interesse, die auf Basis einer formal definierten Notation erstellt wurden. Beispiele für solche Artefakte sind einzelne Strukturdiagramme, Automaten oder Zustandsübergangstabellen. Die von einem Artefakt verwendete Notation wird durch seinen *Artefakttyp* festgelegt.

Definition 2.1.5 (Artefakttyp)
Ein *Artefakttyp* besteht aus

- einem eindeutigen Bezeichner und
- einer Notation. ❍

Prinzipiell können verschiedenen Artefakttypen dieselbe Notation verwenden und sich lediglich durch ihren Bezeichner unterschieden. Dies ist dann sinnvoll, wenn dieselbe Notation verwendet wird, um verschiedene Aspekte eines Systems oder Eigenschaften auf verschiedenen Abstraktionsebenen zu beschreiben. So können beispielsweise Sequenzdiagramme zum einen verwendet werden, um auf abstrakter Ebene Abläufe zwischen Akteuren einer Anwendungsdomäne zu modellieren oder aber um detailliert die Kommunikation von Komponenten eines Systems zu spezifizieren. Um deutlich zu machen, dass Artefakte in dieser Notation jeweils unterschiedlich zu interpretieren sind, wird ihnen trotz gleicher Notation jeweils ein unterschiedlicher Artefakttyp zugeordnet.

Die Menge aller zu einem Zeitpunkt erschaffenen Artefakte und ihre Beziehungen bilden ein *Modell der Artefakte*. Durch ein *Produktmodell* wird festgelegt, wie solche Modelle innerhalb eines konkreten Entwicklungsprozesses strukturiert sein dürfen.

Definition 2.1.6 (Produktmodell, Modell der Artefakte)
Ein *Produktmodell* bezeichnet ein Metamodell, welches die Typen der in einem Entwicklungsprozess erzeugten Entwicklungsartefakte und ihre möglichen Beziehungen untereinander festlegt.

Die Instanz eines Produktmodells ist ein *Modell der Artefakte* in einem konkretem Entwicklungsprojekt. ❍

Im Verlauf eines Entwicklungsvorhabens wird eine Instanz des Produktmodells sukzessiv aufgebaut und erweitert. Hierbei muss sichergestellt sein, dass dieses Modell der Artefakte zu jedem Zeitpunkt konsistent zu seinem Produktmodell ist.

Aktivitäten bezeichnen Entwicklungsschritte, die von Projektbeteiligten vorgenommen werden. In der *Beschreibung einer Aktivität* wird festgelegt, wie ausgehend von einer Menge von Ausgangsartefakten neue Artefakte produziert oder bestehende verändert werden. Die Dokumentation der Entwicklungsschritte erfolgt hierbei zumeist in Form eines für den Entwickler verständlichen Textes.

Definition 2.1.7 (Aktivitätsbeschreibung)
Eine *Aktivitätsbeschreibung* definiert, wie Artefakte manipuliert und erzeugt werden können. Hierzu umfasst die Definition einer Aktivität

- die Spezifikation einer Abbildung von einer Menge von Ausgangsartefakten auf eine Menge von Zielartefakten und
- eine Dokumentation des Vorgehens zur Durchführung der Aktivität. Diese kann auch Verweise auf durchzuführende Unteraktivitäten enthalten. ❍

Das Produktmodell und die Beschreibungen der Aktivitäten eines Entwicklungsprozesses bilden zusammen mit einer Vorgabe, wie die Aktivitäten im Verlauf der Entwicklung nacheinander ausgeführt werden, ein *Prozessmodell*.

Definition 2.1.8 (Prozessmodell)
Ein *Prozessmodell* definiert eine Klasse von konkreten Entwicklungsprozessen. Hierzu umfasst es

- ein Produktmodell,
- eine Menge von Aktivitätsbeschreibungen,
- einen Mechanismus der mögliche Abfolgen von Aktivitäten im Verlauf eines Entwicklungsprozesses festlegt. ❍

Für die Beschreibung von Prozessmodellen werden wiederum geeignete Metamodelle benötigt. Mittlerweile existiert eine große Anzahl von Ansätzen und Werkzeugen zur Modellierung und Analyse von Entwicklungsprozessen [BFG93, SCJD01, FKN94]. In Abschnitt 2.2.2 wird ein eigener Ansatz zur Modellierung von Entwicklungsprozessen vorgestellt, der es im Gegensatz zu den existierenden Techniken erlaubt, bestehende Prozessmodelle zu integrieren und Aussagen über die Durchführbarkeit von Entwicklungsschritten in einer gegebenen Projektsituation zu machen.

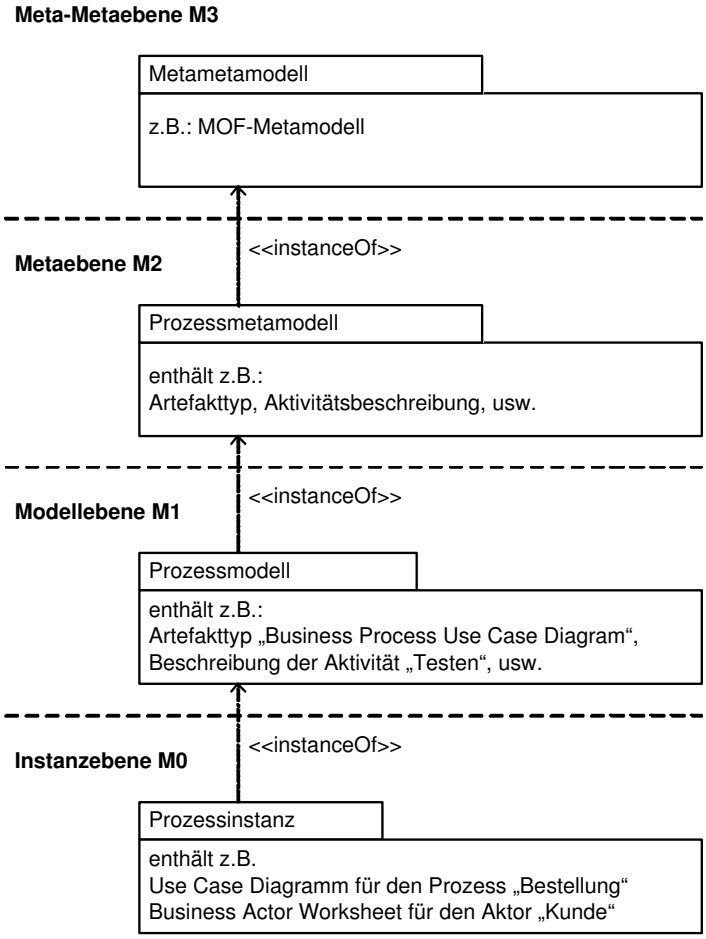


Abbildung 2.2: Die MOF Metamebenen und das Metamodell für Entwicklungsprozesse

Abbildung 2.2 skizziert die verschiedenen Metaebenen von Prozessmodellen. Auf der obersten Ebene findet sich ein Metametamodell, das verwendet wird, um ein Prozessmetamodell zu definieren. Hierzu kann beispielsweise das MOF-Metamodell verwendet werden. Ein Prozessmetamodell dient als Sprache zur Spezifikation von Prozessmodellen. Ein Beispiel für ein Prozessmodell ist der in Abschnitt 5.1 vorgestellten KOGITO-Prozess. Instanzen eines Prozessmodells sind Prozesse in Ausführung, d.h. ein konkretes Entwicklungsprojekt in seiner Ausführung und die Menge der im Verlauf der Projektdurchführung erzeugten Artefakte.

Vergleicht man diese Abbildung mit den Metaebenen der MOF (siehe Abbildung 1.1, S. 7), die sich an konzeptuellen Modellen anstelle von Prozessmodellen orientiert, so stellt man fest, dass ein Entwickler in einem Projekt mit Modellen zur Beschreibung von Software-Systemen der Ebene M1 arbeitet. Der Prozess, dessen Bestandteil die Erschaffung dieser konzeptuellen Modelle ist, findet sich

jedoch auf der Ebene M0 der Entwicklungsprozesse, da es sich jeweils um einen konkreten Entwicklungsprozess in Ausführung handelt.

2.1.3 Modelle zur Beschreibung von Software-Systemen

Im Rahmen dieses Abschnitts werden Modelle betrachtet, welche der Beschreibung eines zu schaffenden Software-Systems und dessen Umgebung dienen. Zunächst stellt jedes in einem Entwicklungsprozess geschaffene Artefakt (vgl. vorangegangener Abschnitt) eine Beschreibung eines Teils oder Aspektes des Gesamtsystems dar. Die von einem Artefakttyp verwendete Syntax wird jeweils durch eine *Notation* festgelegt.

Definition 2.1.9 (Notation)
Eine *Notation* besteht aus

- einem eindeutigen Bezeichner *id*,
- einem Metamodell $mm \in \overline{\mathbb{MM}}$, welches die *abstrakte Syntax* der Notation festlegt,
- einer Spezifikation der *konkreten Syntax* der Notation und
- einer Spezifikation der Abbildung der konkreten Syntax auf die abstrakte Syntax. ❍

Mögliche Notationen umfassen zum einen textuelle Beschreibungstechniken, wie z.B. Tabellen oder strukturierter Text. Zum anderen existiert eine Vielzahl von graphischen Notationen, wie beispielsweise die verschiedenen Notationen der UML.

Die abstrakte Syntax einer Notation beschreibt, aus welchen Arten von Elementen (z.B. Zustände und Transitionen) sich die Notation zusammensetzt und wie diese Elemente ineinander in Bezug stehen können. Innerhalb der abstrakten Syntax eines Automaten wird beispielsweise festgelegt, dass jeder Zustandsübergang eines Automaten genau zwei, nicht notwendiger Weise verschiedene, Zustände verbindet.

Das Aussehen der Elemente einer Notation und ihre mögliche graphische Anordnung wird durch ihre konkrete Syntax festgelegt. Diese kann ebenfalls durch ein geeignetes Metamodell spezifiziert sein, jedoch sind auch informellere Spezifikationen in Form von Text und Graphiken denkbar. Dementsprechend kann auch die Abbildung auf die abstrakte Syntax der Notation informellen Charakter haben.

Ein zu erstellendes Software-System wird durch ein *konzeptuelles Modell* beschrieben. In einem solchen Modell finden sich alle Informationen aus den im Entwicklungsprozess erstellten Artefakten. Die in einem konzeptuellen Modell verwendeten Konzepte zur Modellierung von Software-Systemen und ihre möglichen Beziehungen untereinander werden in einem *konzeptuellen Metamodell* definiert. Typische Konzepte zur wie sie bei der Modellierung von Software-Systemen verwendet werden sind beispielsweise die „Komponente“, „Zustand“ oder „Zustandsübergang“.

Definition 2.1.10 (Konzeptuelles Modell)
Ein *konzeptuelles Modell* ist ein Modell eines zu erstellenden Systems und ggf. den Teilen seiner Umwelt, die für die Realisierung des Systems relevant sind.
Metamodelle konzeptueller Modelle werden *konzeptuelle Metamodelle* genannt. ❍

Ein konzeptuelles Metamodell definiert also eine Menge gültiger konzeptueller Modelle. Es legt fest, welche Instanzen von Konzepten ein konzeptuelles Modell enthalten darf und welche Beziehungen

zwischen diesen Instanzen bestehen dürfen. Durch die Festlegung auf ein konzeptuelles Metamodell wird für alle seine Instanzen ein einheitlicher Abstraktionsgrad bestimmt.

Durch Artefakte, bzw. die Modelle der abstrakten Syntax ihrer Notation, werden jeweils Teile des Gesamtsystems beschrieben. Diese Ausschnitte von konzeptuellen Modellen, die zumeist einen Teilaspekt eines solchen Modells umfassen, werden im weiteren als *Sichten* bezeichnet. Oftmals lassen sich verschiedene Sichten aus unterschiedlichen Arten von Artefakten gewinnen.

Definition 2.1.11 (Sicht)
Eine *Sicht* ist ein Ausschnitt eines konzeptuellen Modells. Die Art des Ausschnitts wird durch einen *Aspekt* festgelegt, der definiert, welche Teile des konzeptuellen Modells in der entsprechenden Sicht enthalten sind. ❍

Ein typischer Aspekt in der Software-Entwicklung ist beispielsweise die Struktur eines Software-Systems. Eine zu diesem Aspekt gehörige Sicht würde dementsprechend alle Teile eines konzeptuellen Modells umfassen, welche die Struktur des Systems festlegen. Teile des konzeptuellen Modells, welche das Systemverhalten charakterisieren, sind hingegen nicht in dieser Sicht enthalten. Eine Beispiel für eine Notation zur Beschreibung einer Struktursicht sind Komponentendiagramme.

Um die Konsistenz einer Systemspezifikation gewährleisten zu können, muss die Konsistenz der verschiedenen Sichten untereinander sichergestellt werden. Hierzu existieren je nach der Art der verwendeten Modelle unterschiedliche Ansätze. So wird beispielsweise in [FGH+94] die Konsistenz einer Menge von Artefakten, bzw. deren Sichten durch den paarweisen Vergleich der einzelnen Sichten verifiziert.

Eine anderer Ansatz ist es, die Sichten zu einem Gesamtmodell zu integrieren. Dies kann durch die Konjunktion einer Menge prädekatenlogischer Aussagen oder aber durch die Integration objektorientierter Modelle zu einem Gesamtmodell, wie im Fall der UML, erfolgen. Innerhalb des folgenden Abschnitts werden Ansätze zu einer flexiblen Integration von Artefakten in ein gemeinsames konzeptuelles Modell diskutiert.

2.2 Modelltransformationen in der Software-Entwicklung

Im Rahmen eines Ansatzes zur modellbasierten Entwicklung werden in der Regel eine Vielzahl unterschiedlicher Arten von Modellen verwendet, um ein Software-System bzw. den Entwicklungsprozess zu seiner Entstehung zu modellieren. Innerhalb dieses Abschnitts wird skizziert, in welcher Weise ein solcher Ansatz von geeigneten Mechanismen zur Transformation von Modellen profitieren kann. Die hier angestellten Überlegungen beschränken sich nicht auf objektorientierte Modelle, für deren Transformation im weiteren Verlauf der Arbeit Lösungsansätze erarbeitet werden.

Abbildung 2.3 skizziert exemplarisch die verschiedenen Anwendungsfälle für Modelltransformationen innerhalb eines modellbasierten Entwicklungsprozesses. Im einzelnen sind dies:

- Ein *Entwicklungsprozess* lässt sich in Form von Transformationen des Modells der Artefakte definieren. Durch die Spezifikation der Transformation wird festgelegt, wie aus einer Menge von Ausgangsartefakten neue Artefakte erschaffen oder bestehende manipuliert werden.
- Die *Integration* von Sichten, die durch unterschiedliche Arten von Artefakten gebildete werden, in ein einheitliches konzeptuelles Modell ist eine weitere Aufgabe, welche durch die Transformation von Modellen flexibel gelöst werden kann. Sind die Artefakte mit Hilfe einer wohldefinierten Beschreibungstechnik verfasst, so existiert jeweils eine Transformationsspezifikation die angibt, wie sich deren Inhalte in ein konzeptuelles Modell integrieren lassen.

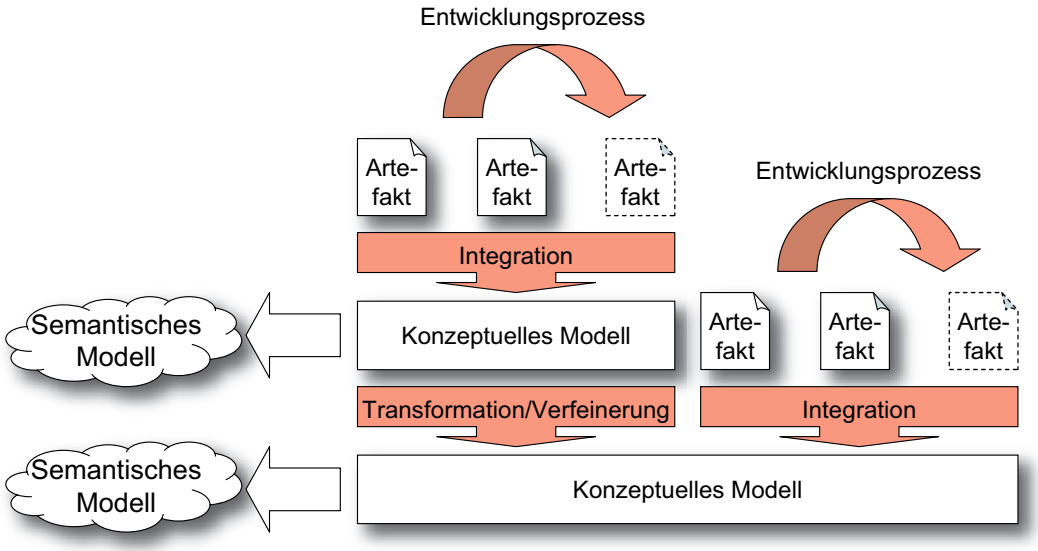


Abbildung 2.3: Modelltransformationen innerhalb eines modellbasierten Entwicklungsprozesses

- Abbildungen zwischen Modellen erlauben eine (automatisierte) *Verfeinerung* abstrakter konzeptueller Modelle. Dies ermöglicht es beispielsweise, abstrakte konzeptuelle Modelle für unterschiedliche technische Implementierungsplattformen wiederzuverwenden. Die so verfeinerten konzeptuellen Modelle können anschließend, wie in der Abbildung angedeutet, durch die Integration zusätzlicher Artefakte erweitert werden.

- Durch eine *semantische Abbildung* werden konzeptuelle Modelle auf semantische Modelle abgebildet. Anhand semantischer Modelle lassen sich auf Basis des gewählten Kalküls Eigenschaften eines modellierten Software-Systems nachweisen. Ein Beispiel hierfür ist die Verifikation von Lebendigkeitseigenschaften von Petrinetzen [Rei82].

Da der Schwerpunkt dieser Arbeit auf der Transformation objektorientierter Modelle liegt, werden vorwiegend die ersten drei Verwendungsformen für Modelltransformationen behandelt werden.

In den folgenden Abschnitten werden zunächst mögliche Eigenschaften von Spezifikationssprachen für Modelltransformationen vorgestellt, die eine Klassifikation solcher Sprachen ermöglicht. Im Anschluss erfolgt eine Diskussion der Anforderungen an eine solche Sprache für die Transformation von Instanzen von Produktmodellen, die Verfeinerung konzeptueller Modelle und die Integration von Artefakten in ein konzeptuelles Modell.

2.2.1 Klassifikation von Spezifikationssprachen für Modelltransformationen

Im Folgenden wird eine Reihe von Eigenschaften von Spezifikationssprachen und Mechanismen zur Modelltransformation angeführt, anhand derer sich verschiedene Lösungen hierfür klassifizieren lassen. Ähnliche Ansätze zur Klassifikation von Modelltransformationssprachen finden sich auch in [CH03] und [GGKH03].

Art der transformierten Modelle: Diese Eigenschaft bezeichnet den Typ der zu transformierenden Modelle. Dieser wird für ein gegebenes Metametamodell *mm* durch die Menge $\mathbb{M}^2_{mm}$ spezifiziert. Mögliche Ausprägungen sind z.B. Graphen (z.B. bei Graphgrammatiken) oder MOF-

Instanzmodelle (wie dies z.B. bei QVT-Transformationen der Fall ist). Ggf. lassen sich verschieden Modelltypen aufeinander abbilden.

Kardinalität der Ein-/Ausgabemodelle: Diese Eigenschaft bezieht sich auf die mögliche Anzahl von Ein- und Ausgabemodellen, die ein Transformationsmechanismus gleichzeitig verarbeiten kann. Im einfachsten Fall wird ein bestehendes Modell verändert (Rewrite-Transformationen) oder ein Quellmodell in ein Zielmodell überführt. Denkbar ist auch die Integration von von mehreren Quellmodellen in ein Zielmodell, die Erzeugung von mehreren Zielmodellen aus einem Quellmodell oder auch aus mehreren Quellmodellen.

Verwendetes Sprachparadigma: Hier kann im wesentlichen zwischen deklarativen, imperativen und hybriden Ansätze zur Spezifikation von Transformationen unterschieden werden. Die Ausprägung der jeweils verwendeten Sprache zur Spezifikation von Modelltransformationen hat einerseits großen Einfluss auf die Mächtigkeit und die Eigenschaften des zugrundeliegenden Transformationsmechanismus und andererseits auf die Benutzerfreundlichkeit des jeweiligen Ansatzes. Deklarative Sprachen umfassen sowohl funktionale als auch logische Sprachen. Kennzeichen deklarativer Sprachen ist die Beschreibung eines Zustandes. Demgegenüber beschreiben imperative Sprachen einen Ablauf. Hybride Ansätze kombinieren deklarative und imperative Sprachkonstrukte zur Spezifikation von Modelltransformationen.

Verarbeitung primitiver Typen: Eine Vielzahl von Arten von Modellen, wie z.B. objektorientierte Modelle, setzen sich aus komplexen Strukturen (Objektgeflechten) und Daten primitiven Typs (z.B. `int`-Werten) zusammen. Während Strukturen durch Modelltransformationen für solche Arten von Modellen immer verarbeitet werden, ist eine Unterstützung für die Abbildung von Werten primitiven Typs nicht immer gegeben. Oftmals wird lediglich das Kopieren solcher Werte unterstützt.

Bijektivität: Dies bezeichnet die Fähigkeit eines Transformationsmechanismus Modelle bijektiv ineinander abbilden zu können. Dementsprechend erhält man durch zwei entsprechend nacheinander ausgeführten Hin- und Rücktransformationen wieder das ursprüngliche Quellmodell als Ergebnis. Hierbei sind die folgenden Ausprägungen denkbar:

- **Generell bijektive Abbildungen:** Jede Transformation ist bijektiv in jeweils eine Hin- und eine Rückrichtung ausführbar.
- **Nachweisbar bijektive Abbildungen:** Es existiert eine Klasse von Transformationen, für die nachgewiesen werden kann, dass sie immer bijektiv ausführbar sind.
- **Inverse Transformationsspezifikationen:** Zu allen oder einer Klasse von Transformationsspezifikationen können inverse Transformationsspezifikationen generiert werden.

Rückverfolgbarkeit: Die Rückverfolgbarkeit einer ist Transformation gegeben, falls für Elemente des Zielmodells ermittelt werden kann, aus welchen Elementen des Quellmodells bzw. der Quellmodelle sie erzeugt wurden.

Kopplung der Modelle: Die durch eine Modelltransformation erzeugte Modelle können von ihren Quellmodellen *abhängig* oder *unabhängig* sein. Sind Zielmodelle von ihren Quellmodellen abhängig, so werden Änderungen im Ursprungsmodell automatisch an das Zielmodell propagiert. In diesem Fall spricht man von *inkrementellen Transformationen*. Weiterhin können Modelle *einfach* oder *bidirektional* gekoppelt werden. Bei der bidirektionalen Kopplung von Modellen werden auch Änderungen im Zielmodell an das Quellmodell propagiert.

Korrektheit von Spezifikationen: Für die Spezifikation einer Modelltransformation sollte sichergestellt sein, dass sie

- ausführbar ist und
- ihre Ausführung Modelle erzeugt, die konform zu einem gegebenen Zielmetamodell sind.

Für die Gewährleistung dieser Eigenschaften gibt es die folgenden Möglichkeiten:

- Die Eigenschaften sind durch die Syntax der Sprache gewährleistet, d.h. jede syntaktisch korrekte Spezifikation ist ausführbar und führt ggf. zu einem Modell, welches eine Instanz eines Zielmetamodells ist.
- Die Eigenschaften können für eine gegebene Spezifikation nachgewiesen werden.
- Ein Nachweis, dass eine Spezifikation über diese Eigenschaften verfügt ist generell nicht möglich.

Fehlerverhalten: Ist die Korrektheit von Transformationsspezifikationen nicht bereits durch die Syntax der verwendeten Sprache sichergestellt, so können im Verlauf einer Modelltransformation ggf. Fehler auftreten, da das erzeugte Zielmodell nicht dem Zielmetamodell entspricht oder für ein Quellmodell kein eindeutiges Zielmodell erzeugt werden kann. Auf solche Fehler gibt es eine Reihe möglicher Reaktionen:

- Abbruch der Transformation
- Ignorieren der Fehler
- Ignorieren der Fehler und Aufsammeln der Fehlerinformation
- Transaktionsgeschützte Transformationen, d.h. Teile der Transformation verlaufen entweder fehlerfrei oder sie erzeugen keine Ausgabe

Mächtigkeit: Modelltransformationssprachen unterschieden sich bezüglich der Mächtigkeit der Menge der möglichen Abbildungen, welche durch sie spezifiziert werden können. Oftmals ist die Mächtigkeit durch die Art des verwendeten Sprachparadigmas zur Spezifikation beschränkt. Während imperative Ansätze über eine sehr große Ausdrucksmächtigkeit verfügen, weisen regelbasierte Ansätze wie beispielsweise Graphgrammatiken oftmals Beschränkungen bei der Berücksichtigung konvexer Hüllen oder der Nicht-Existenz von Modellelementen auf.

Anwenderfreundlichkeit: Eine wesentliches Kriterium für die Benutzerakzeptanz einer Sprache zur Spezifikation von Modelltransformationen ist deren Anwenderfreundlichkeit. Hierbei spielen in erster Linie Eigenschaften des jeweils eingesetzten Sprachparadigmas eine Rolle. Im einzelnen lässt sich die Anwenderfreundlichkeit eines Ansatzes in die folgenden Teilaspekte untergliedern:

Lesbarkeit: Generell sind graphische Notationen für den Anwender besser verständlich. Die Lesbarkeit einer Spezifikation wird weiterhin durch die Verwendung bekannter und akzeptierter Konzepte und Sprachelemente (wie z.B. solche der UML-Notationen) erhöht. Auch die Art der Verknüpfung von graphischen und algorithmischen Sprachelementen hat einen deutlichen Einfluss auf die Lesbarkeit von Spezifikationen.

Verständlichkeit: Neben einer gut lesbaren Syntax sollte der zugrundeliegende Mechanismus zur Modelltransformation möglichst intuitiv verständlich sein.

Abstraktion: Die Möglichkeit Transformationsspezifikationen auf bestimmte Aspekte einzuschränken oder sie abstrakt zu formulieren erlaubt es dem Entwickler, einen besseren Überblick über komplexe Transformationen zu behalten.

Komponierbarkeit: Eine weitere Eigenschaft die den Umgang mit Transformationsspezifikationen erleichtert ist deren Komponierbarkeit. Lassen sich Transformationsspezifikationen aus verschiedenen, bestehenden Spezifikationen zusammensetzen, so erleichtert dies deren Wiederverwendung und die verteilte Entwicklung von Spezifikationen.

Ausführbarkeit: Transformationen müssen automatisiert ausführbar sein. Dies erfordert insbesonders eine hinreichend formale Spezifikationssprache, die es ggf. erlaubt, den Code zur Ausführung der Transformationen zu generieren oder eine Spezifikation durch einen Interpreter zur Laufzeit auszuführen.

Effizienz: Transformationen sollten zur Laufzeit möglichst effizient ausführbar sein.

Realisierbarkeit: Um in der Praxis einsetzbar zu sein, sollte für einen Mechanismus zur Modelltransformation eine Werkzeugunterstützung mit vertretbarem Aufwand realisierbar sein.

Erweiterbarkeit: Erweiterbare Ansätze zur Modelltransformation erlauben es, den bestehenden Mechanismus an spezielle Problemfelder anzupassen. Denkbar sind hier verkürzte Notationen, die eine vereinfachte Spezifikation von Transformation von Modellen eines bestimmten Typs ermöglichen. Ein Beispiel hierfür wäre eine angepasste Notation für die Spezifikation von Transformationen von Automatenmodellen. Diese könnte anstelle von Elementen der abstrakten Syntax von Automaten solche der konkrete Syntax (also Zustände und Übergänge) verwenden.

2.2.2 Der Prozessmusteransatz

Mittlerweile stehen eine Vielzahl von Vorgehensweisen und Methodiken für das Software- und Systems-Engineering zur Verfügung, die jeweils für verschiedene Arten von Projekten mehr oder weniger gut geeignet sind. So eignet sich der Ansatz des eXtreme Programming [Bec99] generell besser für kleinere Projekte, während das V-Modell [BD93] eine sinnvolle Vorgehensweise für Projekte größeren Umfangs anbietet.

Ein Merkmal, welches vielen heutigen Software-Entwicklungsprojekten gemein ist, ist die Notwendigkeit zur Kombination der jeweils besten Elemente unterschiedlicher Ansätze, um den vielfältigen Anforderungen an einen optimierten Entwicklungsprozess gerecht zu werden. Bestehende Vorgehensmodelle wie der Objectory Process [JCJO92], der Rational Unified Process [Kru00a], Catalysis [DW98], oder das V-Modell erlauben es, den Entwicklungsprozess vor der Projektdurchführung an die jeweiligen Erfordernisse eines Projektes anzupassen. Dieses *Tailoring* ist jedoch nur in begrenztem Umfang möglich. Eine Kombination verschiedener Vorgehensweisen ist nicht vorgesehen.

Ein weiteres Merkmal heutiger Software-Entwicklungsvorhaben ist das sich in zunehmendem Maße schnell ändernde Umfeld. Dieses ist beispielsweise durch Änderungen in der Anwendungsdomäne, der Organisationsstruktur der beteiligten Partner oder aber des technischen Umfeldes bedingt. Dieser Umstand macht es notwendig den Entwicklungsprozess auch während der Projektdurchführung an die neuen Erfordernisse anpassen zu können [Wei97]. Bestehende Vorgehensmodelle lassen die hierfür notwendige Flexibilität vermissen, ein sog. *dynamisches Tailoring* des Entwicklungsprozesses, also die Anpassung des Entwicklungsprozesses während eines laufenden Projektes, wird nicht unterstützt.

Der Grund für die mangelnde Integrierbarkeit bestehender Vorgehensweisen und den Mangel an Flexibilität bei der Anpassung an ein sich änderndes Umfeld ist einerseits das Fehlen einer einheitlichen Sprache zur Definition von Vorgehensmodellen. Andererseits sind praktisch alle existierenden

Vorgehensmodelle aktivätszentriert, d.h. ein Entwicklungsprozess wird als eine Folge von Aktivitäten definiert, in deren Verlauf Dokumente erzeugt und verarbeitet werden.

Um diese Mängel zu beheben wird an dieser Stelle ein Ansatz zur Spezifikation von Prozessmodellen mit Hilfe sogenannter Prozessmuster eingeführt. Der hier vorgestellte Prozessmusteransatz wurde im Rahmen des Teilprojekts ZEN des Forschungsverbundes FORSOFT II der Bayerischen Forschungsstiftung entwickelt [GMP+03b, Mic03, GMP+02a, GMP+02b, GMP+01b, GMP+01a]. Er erlaubt es, Vorgehensweisen in der Software-Entwicklung in einer einheitlichen Weise zu dokumentieren und flexibel miteinander zu kombinieren. Hierzu verfügt der Prozessmusteransatz über ein explizites Metamodell zur Modellierung von Prozessmodellen für die Software-Entwicklung (siehe auch Abbildung 2.2, S. 27). Das Prozessmetamodell besteht im wesentlichen aus zwei Teilen: dem *Produktmetamodell* und dem *Aktivitätsmetamodell*.

Eine Instanz des Produktmetamodells beschreibt die Menge der möglichen Strukturen des im Verlauf eines Entwicklungsprozesses zu schaffenden Geflechts aus Entwicklungsartefakten und den Beziehungen zwischen diesen. Artefakttypen beschreiben im Produktmodell die möglichen Arten von Dokumenten, die im Verlauf des Prozesses geschaffen werden (siehe Definition 2.1.5), während Assoziationen zwischen ihnen mögliche Beziehungen zwischen verschiedenen Artefakten modellieren. Beispielsweise existiert in dem in Abschnitt 5.1 (Seite 179 ff) vorgestelltem KOGITO-Produktmetamodell eine Assoziation zwischen dem Artefakttypen „Business Process Use Case Diagramm“ und „Business Process Description Worksheet“, also den Formularen, welche die im Anwendungsfalldiagramm vorkommenden Geschäftsprozesse näher beschreiben.

Das Aktivitätsmetamodell enthält alle notwendigen Elemente für die Beschreibung von Entwicklungsschritten innerhalb eines Entwicklungsprozesses. Ein Entwicklungsschritt lässt als Transformation einer Instanz des Produktmodells dokumentieren. Die Elemente einer Produktmodellinstanz, welche durch einen Entwicklungsschritt verwendet bzw. erzeugt werden, werden in [GMP+03b] als *Kontext* der Aktivität bezeichnet. Darüber hinaus ermöglicht es das Aktivitätsmetamodell kausale Zusammenhänge und hierarchische Beziehungen zwischen den einzelnen Entwicklungsschritten zu definieren.

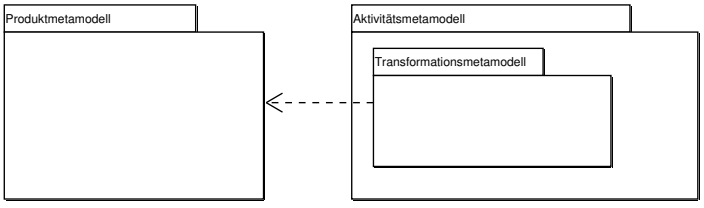


Abbildung 2.4: Zusammenhang zwischen Produktmetamodell und Aktivitätsmetamodell im Prozessmusteransatz

Abbildung 2.4 skizziert den Zusammenhang zwischen Produktmetamodell, Aktivitätsmetamodell und Transformationsspezifikationen.

Produktmodelle

Ein Produktmodell bildet die gemeinsame Basis für die Integration unterschiedlicher Vorgehensweisen innerhalb eines Prozessmodells. Ein Produktmodell legt die mögliche Struktur der im Verlauf des Entwicklungsprozesses geschaffenen Artefakte und deren Beziehungen untereinander fest.

Innerhalb des Prozessmusteransatzes werden Produktmodelle in Form objektorientierter Klassenmodelle spezifiziert. In Abschnitt 3.1 (S. 51ff) wird eine formal fundierte Definition für solche Mo-

delle vorgestellt. Eine Klasse des Produktmodells modelliert jeweils einen Artefakttypen, Assoziationen zwischen diesen Klassen mögliche Beziehungen zwischen den Artefakten. Klassen des Produktmodells können zusätzlich mit Attributen versehen sein, in denen Informationen über den aktuellen Zustand eines Artefakts, wie z.B. „in Bearbeitung“, „gereviewed“, etc, gehalten wird. Dieser Mechanismus kann auch genutzt werden um einzelne Artefakte als gelöscht bzw. ungültig zu markieren.

Ein Modell der Artefakte eines konkreten Entwicklungsprojekts kann dementsprechend durch ein Objektmodell aus den Instanzen des Produktmodells beschrieben werden.

Aktivitätsmodelle

Innerhalb eines Aktivitätsmodells werden zwei mögliche Arten von Vorgehensschritten unerschieden: *Aktivitäten* und *Prozessmuster*.

Eine *Aktivität* spezifiziert einen Entwicklungsschritt ohne Vorgaben über die Art seiner Ausführung zu machen. Dementsprechend legt eine Aktivität lediglich fest, was das Ziel ihrer Ausführung ist, welche Artefakte eines Produktmodells hierfür notwendig sind und welche Artefakte durch sie erzeugt bzw. manipuliert werden. Dementsprechend wird durch eine Aktivität eine Transformation auf Instanzen des Produktmodells spezifiziert. Der Teil des Instanzmodells, der zur Ausführung der Aktivität notwendig ist, wird als *initialer Kontext*, das Ergebnis der Ausführung als *Ergebniskontext* bezeichnet.

Beispielsweise legt eine Aktivität „Create Behavioral Specification“ fest, dass zu einer Systemkomponente, welche in einem Strukturdiagramm einer Anwendung vorkommt, ein Artefakt zur Beschreibung des Verhaltens der Komponente erstellt werden soll. Eine Beschreibung des detaillierten Vorgehens, wie eine solche Spezifikation zu erstellen ist, ist jedoch *nicht* Teil der Aktivitätsbeschreibung.

Prozessmuster enthalten darüber hinaus Beschreibungen *wie* Aktivitäten auszuführen sind. So kann für die Aktivität, wie z.B. die Aktivität „Create Behavioral Specification“, eine Reihe alternativer Prozessmuster existieren, in denen jeweils unterschiedliche Ansätze für das Erstellen einer Verhaltensspezifikation dokumentiert sind.

Ebenso wie Aktivitäten, enthalten Prozessmuster die Spezifikation einer Transformation, in der festgelegt ist, in welcher ein Modell der Artefakte durch die Anwendung des Prozessmusters verändert wird. Damit ein Prozessmuster eine Aktivität realisieren kann, darf der initiale Kontext des Prozessmusters höchstens die Elemente des initialen Kontextes der zu realisierenden Aktivität umfassen. Zugleich muss der Ergebniskontext des Prozessmusters zumindest den in der realisierten Aktivität erzeugten enthalten. Dies entspricht der Aussage, dass das Prozessmuster nicht mehr Artefakte als die von der realisierten Aktivität geforderten Artefakte zu seiner Realisierung benötigt. Zugleich muss es mindestens die in der Aktivität beschriebenen Ausgabeartefakte erzeugen.

Um Abläufe innerhalb eines Prozessmusters modellieren zu können, umfasst dieses weiterhin eine Reihe von *Aktionen*, die über Transitionen verbunden sind und so den gewünschten Ablauf festlegen. Es existiert eine Menge besonderer Aktionen zur Parallelisierung von Teilabläufen, für die Auswahl alternativer Folgetransitionen und zur Kennzeichnung eines Start- bzw. von Endzuständen. Zusätzlich kann jede Transition mit einer Bedingung versehen werden, welche erfüllt sein muss, um die nachfolgende Aktion ausführen zu können. Ein so modellierter Ablauf lässt sich graphisch als UML-Aktivitätsdiagramm darstellen. Eine Aktion verweist entweder auf eine textuelle Beschreibung, in der für den Entwickler verständlich Form dargelegt ist, welche Schritte auszuführen sind, oder aber auf eine zu realisierende Aktivität.

Verweist ein Prozessmuster auf eine für seine Durchführung auszuführende Aktivität, so können an dieser Stelle wieder verschiedene alternative Prozessmuster zu deren Realisierung zur Verfügung

stehen. Dies erlaubt es, innerhalb der Ausführung eines Prozessmusters Handlungsalternativen für Teilaufgaben anzubieten.

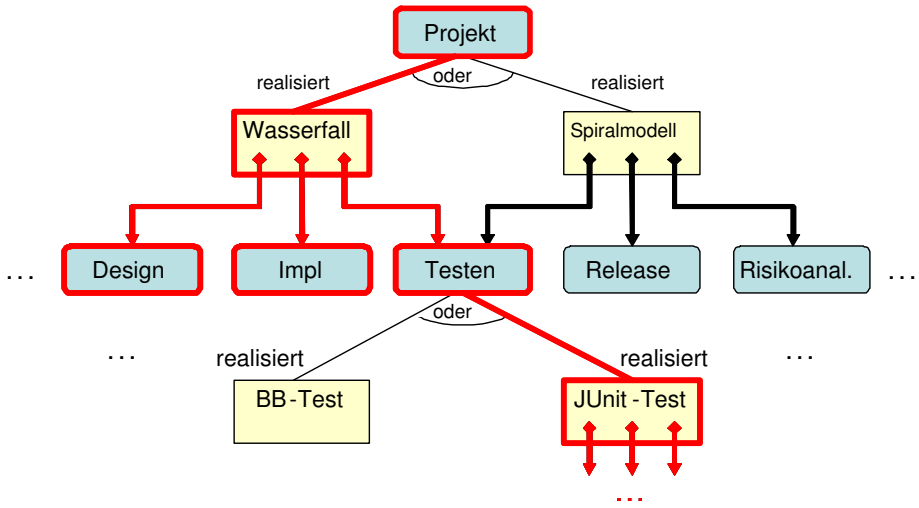


Abbildung 2.5: Beispielhafte Darstellung der Beziehungen zwischen Prozessmustern und Aktivitäten

Abbildung 2.5 verdeutlicht diesen Zusammenhang anhand eines einfachen Beispielszenarios. Aktivitäten sind hierbei als Rechtecke mit runden Ecken, Prozessmuster als eckige Rechtecke, dargestellt. Eine Linie zwischen einer Aktivität und einem Prozessmuster weißt darauf hin, dass das Prozessmuster die entsprechende Aktivität realisiert, während ein Pfeil zwischen zwei Elementen dieses Typs andeutet, dass ein Prozessmuster im Verlauf seiner Ausführung die jeweilige Aktivität ausführt.

Die Aktivität „Projekt“ in Abbildung 2.5 steht stellvertretend für sämtliche Teilaktivitäten die im Rahmen eines Projektes durchzuführen sind. Für die Realisierung dieser Aktivität bieten sich im Beispiel zwei Prozessmuster an, in denen jeweils das Wasserfallmodell [Roy70] oder das Spiralmodell [Boe88] als grobgranulare Vorgehensweisen dokumentiert sind. Innerhalb des Wasserfallmodells wird auf eine Reihe von Subaktivitäten, die zur Durchführung des Prozessmusters ausgeführt werden müssen, verwiesen („Design“, „Implementierung“, „Testen“, etc.). Für die Aktivität „Testen“ bieten sich wiederum zwei realisierende Prozessmuster als mögliche Vorgehensweisen an: der Blackbox-Test („BB-Test“) und das Testen mit dem JUnit-Framework („JUnit-Test“). Die dick markierten Aktivitäten und Prozessmuster kennzeichnen einen „Pfad“ durch das Geflecht aus Aktivitäten und Prozessmustern, der die gewählten Alternativen in einem konkreten Projekt symbolisiert.

Die Spezifikation von Prozessmustern erfolgt anhand eines Formulars (siehe auch [Mic03]), welches sich an den Patterntemplates für die Beschreibung von Design-Mustern [GHJV95, BMR+96] orientiert. Die wesentlichen Elemente des Formulars sind:

Name Der Name des Prozessmusters

Realisierte Aktivitäten Verweis auf eine oder mehrere Aktivitäten, die durch das Prozessmuster realisiert werden

Problem Charakterisierung des Problemfeldes, zu dessen Lösung die Anwendung des Prozessmusters beitragen soll

Projektkontext Beschreibung des Projektumfeldes, -typs, in dem das Prozessmuster angewendet werden kann

Produktmodelltransformation Die Spezifikation der Transformation des Produktmodells, die durch die Anwendung des Prozessmusters entsteht

Ablaufdiagramm Ein UML-Aktivitätsdiagramm, in dem der Ablauf, der für die Durchführung des Prozessmusters notwendigen Aktionen, festgelegt ist; einzelne Aktionen des Diagramms können hierbei Aktivitäten des Prozessmodells referenzieren.

Beschreibung von Aktionen Aktionen, die nicht auf Aktivitäten des Prozessmodells verweisen werden in einer für einen Entwickler verständlichen Weise textuell dokumentiert.

Anwendungsbeispiele Beispiele, in denen die Anwendung des Prozessmusters sich als empfehlenswert erwiesen hat

Eine formale Charakterisierung des Projektkontextes, z.B. durch die Zuordnung von Werten zu vordefinierten Attributen wie Projektgröße, Anwendungsdomäne, etc., erlaubt es, Prozessmuster während eines Projektes werkzeugunterstützt, entsprechend ihrer Eignung für die aktuelle Projektsituation, auszuwählen. In [GMP+03a] wurde eine solche Unterstützung bereits prototypisch realisiert, eine Dokumentation der Toolunterstützung findet sich auch in [GMP+02b].

Ansätze Vorgehensschritte in Form von Prozessmustern zu dokumentieren finden sich auch in [Amb98]. Jedoch werden hier die Veränderungen des Modells der Artefakte durch die Anwendung eines Musters lediglich informell charakterisiert. Der hier vorgestellte Ansatz dokumentiert Enwicklungsschritte als Transformationen des Modells der Artefakte. Demzufolge muss ein Mechanismus zur Transformation solcher Modelle eine Reihe von Anforderungen erfüllen. Im einzelnen sind dies:

Art der transformierten Modelle: Es werden Instanzen von Produktmodellen, d.h. Objektmodelle, transformiert.

Kardinalität der Ein-/Ausgabemodelle: Es müssen Rewrite-Transformationen spezifizierbar sein, durch die ein bestehendes Modell schrittweise erweitert wird.

Verwendetes Sprachparadigma: Deklarative Ansätze sind vorzuziehen, für primitive Typen können auch imperative Sprachkonstrukte verwendet werden.

Verarbeitung primitiver Typen: Primitive Typen sollten transformiert werden können. Es sollte möglich sein, sowohl logische als auch arithmetische Operationen anzuwenden. Darüber hinaus müssen auch Zeichenketten verarbeitet werden können.

Bijektivität: Im Falle von Rewrite-Transformationen nicht möglich

Rückverfolgbarkeit: Im Falle von Rewrite-Transformationen nicht möglich

Korrektheit von Spezifikationen: Es muss sichergestellt werden können, dass die Menge der Prozessmuster eines Prozessmodells nie zu inkonsistenten Instanzen des Produktmodells führen kann.

Es muss möglich sein, Aussagen darüber zu treffen, ob eine oder mehrere Aktivitäten bzw. Prozessmuster generell anwendbar sind. Ist dies nicht der Fall, so ist der entsprechende Vorgehensschritt entweder fehlerhaft spezifiziert oder generell für das gewählte Produktmodell nicht anwendbar.

Es muss möglich sein, nachzuweisen, dass durch eine Menge von Teilaktivitäten eines Prozessmusters immer die von dem Prozessmuster definierten Ausgaben erzeugt werden können.

Fehlerverhalten: Die Ausführung von Transformationsschritten, welche zu einem inkonsistenten Modell der Artefakte führen, sollte nicht möglich sein.

Mächtigkeit: Das Auffinden und Erzeugen von Strukturen im Modell muss möglich sein.

Anwenderfreundlichkeit: Die Anwender sind Modellierungsexperten. Die Konzepte der Modellierungssprache sollten daher auch in der Sprache für Transformationsspezifikationen wiederverwendet werden. Spezifikationen sollten deklarativ und frei von undurchsichtigen Kontrollstrukturen sein. Die Kombinierbarkeit von Spezifikationen ist erforderlich, um nachträglich zusätzliche Vorgehensweisen in einen methodischen Baukasten integrieren zu können.

Ausführbarkeit: Eine automatisierte Ausführung ist nicht unbedingt erforderlich. Es sollte jedoch möglich sein, automatisiert zu verifizieren, ob eine Transformation für ein gegebenes Ausgangsmodell durchführbar ist. Dies entspricht der Aussage, ob eine Aktivität bzw. ein Prozessmuster in einer gegebenen Projektsituation anwendbar ist.

Effizienz: Die Effizienz spielt hierbei eine untergeordnete Rolle.

Realisierbarkeit: Eine Werkzeugunterstützung zur Auswahl geeigneter Prozessmuster ist wünschenswert.

Erweiterbarkeit: Zusätzliche Erweiterungen werden nicht benötigt.

In Kapitel 3.6 wird ein Transformationsmechanismus vorgestellt, der diese Anforderungen erfüllt. Kapitel 5.4 zeigt anhand eines Beispiels wie sich mit diesem Mechanismus Prozessmuster und Aktivitäten eindeutig als Transformationen eines Modells der Artefakte spezifizieren lassen und die gewünschten Eigenschaften eines so modellierten Prozessmodells verifiziert werden können.

2.2.3 Transformation von konzeptuellen Modellen

Im Rahmen eines modellbasierten Entwicklungsansatzes bietet es sich an, statt einem universellen konzeptuellen Modell mehrere konzeptuelle Modelle auf verschiedenen Abstraktionsebenen zu erstellen. Durch geeignet spezifizierte Transformationen lassen sich automatisiert konkretere konzeptuelle Modelle aus abstrakten Modellen erzeugen. Dieses Vorgehen ist zwar keine zwingende Voraussetzung für eine modellbasierte Entwicklung, es bietet jedoch eine Reihe von Vorteilen:

- Die einzelnen konzeptuellen Metamodelle sind einfacher und besser beherrschbar als ein universelles Metamodell.
- Für unterschiedliche Anwendungsdomänen und technische Plattformen lassen sich jeweils gezielt angepasste konzeptuelle Metamodelle verwenden.
- Einmal erstellte abstrakte Modelle lassen sich wiederverwenden. So kann ein Modell der Anwendungsdomäne den Ausgangspunkt für die Modellierung verschiedener Systeme innerhalb dieser Domäne bilden.
- Klar definierte Abbildungsvorschriften erlauben es, je nach Bedarf automatisiert verfeinerte Modelle für unterschiedliche technische Plattformen und Leistungsmerkmale eines Software-Systems zu erzeugen.
- Durch automatisierte ausführbare Transformation von Modellen kann die Konsistenz der Modelle untereinander sichergestellt und unnötige Modellierungsfehler vermieden werden.

In dem hier vorgestellten Ansatz werden konzeptuelle Modelle auf unterschiedlichen Abstraktionsebenen im Laufe des Entwicklungsprozesses erstellt und durch Verfeinerungsabbildungen ineinander übergeführt. Ein solches Vorgehen ist im Umfeld der UML auch Gegenstand der Model Driven Architecture der OMG (siehe Abschnitt 1.3.3, Seite 9).

Um Modelle automatisiert aufeinander abzubilden, bzw. sie verfeinern zu können, müssen die hierzu notwendigen Transformationen eindeutig spezifiziert werden. Zum einen muss die Spezifikation einer solchen Transformation sicherstellen, dass die Transformation für ein Ursprungsmodell ein *deterministisches* Ergebnis liefert. Zum anderen muss für eine Transformation gewährleistet sein, dass sie für *jedes beliebige* Quellmodell ein Zielmodell erzeugt, d.h. das Ergebnis der Transformation muss als Modell interpretierbar sein. Dementsprechend ist eine Modelltransformationsspezifikation als eine totale Abbildung zwischen Modellen mit dem selben Metametamodell definiert:

Definition 2.2.1 (Modelltransformationsspezifikation)
Eine *Modelltransformationsspezifikation* oder kurz *Modelltransformation mt* von Modellen mit einem gegeben Metametamodell *mm* ist eine totale Abbildung zwischen Modellen:

$$mt : \overline{\mathbb{M}}^2_{mm} \rightarrow \overline{\mathbb{M}}^2_{mm}$$ ❍

Die Menge $\overline{\mathbb{M}}$ aller denkbaren Modelle ist generell unbeschränkt und umfasst Modelle unterschiedlichster Ausprägung. Folglich kann in der Praxis keine Sprache für Transformationsspezifikationen zwischen *beliebigen* Modellen aus $\overline{\mathbb{M}}$ gefunden werden, welche alle denkbaren Ausprägungen von Modellen berücksichtigt. Aus diesem Grunde beschränkt sich Definition 2.2.1 auf Klassen $\overline{\mathbb{M}}^2_{mm}$ von Modellen mit demselben Metametamodell *mm*. Ohne eine solche Beschränkung wäre die Angabe einer Spezifikationssprache für nicht-triviale Modelltransformationen praktisch unmöglich. Ein Beispiel für eine denkbare triviale Abbildung wäre die Transformation beliebiger Quellmodelle auf die leere Menge.

Für die Menge aller Objektmodelle (unabhängig von ihrem jeweiligen Klassenmodell) wird in Kapitel 3 die Sprache BOTL zur Spezifikation von Modelltransformationen vorgestellt.

Definition 2.2.1 fordert, dass eine durch eine Transformationsspezifikation formulierte Modelltransformation für *beliebige* Quellmodelle aus $\overline{\mathbb{M}}^2_{mm}$ *immer* ein deterministisches Ergebnis in Form eines Modells aus $\overline{\mathbb{M}}^2_{mm}$ erzeugt. Diese Eigenschaft ist bedeutend, da so sichergestellt ist, dass *beliebige* konzeptuelle Modelle durch sie automatisiert verfeinert werden können. Wie sich in Abschnitt 4.1 zeigt, ist diese Forderung im allgemeinen keineswegs trivial und bedarf einer Reihe von durchdachten Verifikationstechniken, um sicherzustellen, dass jedes Ergebnis einer Transformation ein Element aus $\overline{\mathbb{M}}^2_{mm}$ ist.

Wird die Menge der möglichen Quellmodelle durch ein Metamodell mm' weiter eingeschränkt, so entspricht dies der Aussage, dass eine Modelltransformation

$$mt : \overline{\mathbb{M}}_{mm'} \rightarrow \overline{\mathbb{M}}^2_{mm} \qquad \text{mit } mm' \in \overline{\mathbb{M}}_{mm}$$

eine totale Abbildung ist. D.h. die Transformation muss nur Modelle als Ausgaben liefern, falls ihre Eingabe ein Element der durch das Metamodell mm' spezifizierten Menge von Modellen ist.

In der Regel wird von einer Modelltransformation jedoch zusätzlich erwartet, dass sie Modelle erzeugt, die Instanzen eines gegebenen Zielmetamodells sind. Soll beispielsweise durch eine MDA-Transformation ein plattformunabhängiges UML-Modell auf eine Instanz eines UML-CORBA-Profils abgebildet werden, so soll sichergestellt sein, dass jedes so erzeugte Zielmodell auch tatsächlich konform zu dem gegebenem CORBA-Profil ist. Modelltransformationen mit dieser Eigenschaft bezüglich eines Quell- und eines Zielmetamodells werden als *metamodellkonform* bezeichnet.

Definition 2.2.2 (Metamodellkonforme Modelltransformationsspezifikation)
Wird durch das Metamodell mm_0 die Menge der gültigen Quellmodelle und durch mm_1 die der gültigen Zielmodelle definiert, so heißt die Modelltransformationsspezifikation

$$mt : \overline{\mathbb{M}}_{mm_0} \rightarrow \overline{\mathbb{M}}_{mm_1}$$

metamodellkonform, falls mt eine totale Abbildung ist. ❍

Die Transformation eines abstrakten in ein konkreteres Modell ist mit der Anreicherung des abstrakten Modells durch zusätzliche Information verbunden. Je nach Art der gewählten Transformationsspezifikation können Modelle so um zusätzliche Aspekte wie z.B. Sicherheitsmechanismen angereichert werden. Bei der Abbildung von plattformunabhängigen Modelle auf plattformabhängige Modelle, wie sie z.B. der MDA-Ansatz vorsieht, werden konzeptuelle Modelle beispielsweise um die notwendigen Elemente zur Nutzung einer gewählten technischen Infrastruktur erweitert.

Durch die Verfeinerung eines Modells muss die grundlegende Struktur des Ursprungsmodells nicht notwendigerweise erhalten bleiben, jedoch sollte der Informationsgehalt erhalten bleiben. Das Quellmodell einer Verfeinerung kann in diesem Fall als Abstraktion des verfeinerten Modells gesehen werden. Eine Abstraktion eines Modells lässt sich demnach wie folgt definieren:

Definition 2.2.3 (Abstraktion)
Eine *Abstraktion abstract* ist eine *totale*, *surjektive* Abbildung zwischen Modellen, d.h. es gilt:

$$abstract : \overline{\mathbb{M}}_{mm_r} \rightarrow \overline{\mathbb{M}}_{mm_a}$$

wobei mm_r das Metamodell der verfeinerten Modelle und mm_a das Metamodell der abstrakteren Modelle bezeichnet. ❍

Der hier vorgestellte Abstraktionsbegriff ist bewusst sehr allgemein gehalten. In der Praxis werden oftmals gezielt Abstraktionen bezüglich verschiedener Aspekte wie Struktur oder Verhalten verwendet. Dennoch müssen alle diese Abstraktionen der hier vorgestellten Definition genügen. Hierzu wird zunächst ein Äquivalenzbegriff für Modelle eingeführt:

Definition 2.2.4 (Äquivalente konzeptuelle Modelle)
Für ein beliebiges Metamodell mm heißen zwei konzeptuelle Modelle $m_0, m_1 \in \overline{\mathbb{M}}_{mm}$ *äquivalent* (kurz: $m_0 \eqsim m_1$) bezüglich einer Semantik, falls ihre semantische Abbildung jeweils äquivalente semantische Modelle liefert. ❍

Der Äquivalenzbegriff für konzeptuelle Modelle stützt sich also vollständig auf die Äquivalenz der semantischen Interpretation der Modelle ab. Folglich muss ein Kalkül, auf dessen Basis die Semantik konzeptueller Modelle definiert wird, einen entsprechenden Äquivalenzbegriff vorweisen. Aufbauend auf diese Definition lässt sich für eine gegebene Abstraktionsabbilung zwischen zwei Arten von Modellen eine Verfeinerungsrelation definieren, die angibt, ob ein Modell eine Verfeinerung eines anderen Modells bezüglich der gewählten Abstraktion darstellt.

Definition 2.2.5 (Verfeinerung, Verfeinerungsabbildung)
Eine *Verfeinerung* bezüglich einer Abstraktionsabbildung a ist eine Relation $refines_a$ zwischen Modellen für die gilt:

$$refines_a : \mathbb{M}_{mm_r} \times \mathbb{M}_{mm_a} \rightarrow \mathbb{B} \text{ mit}$$
$$\forall m_r \in \mathbb{M}_{mm_r} : refines_a(m_r, m_a) :\Leftrightarrow a(m_r) \eqsim m_a$$

Eine Modelltransformation $refine_a$ heißt *Verfeinerungsabbildung* bezüglich einer Abstraktion a, falls gilt:

$$refine_a : \mathbb{M}_{mm_a} \rightarrow \mathbb{M}_{mm_r} \text{ mit}$$
$$\forall m_a \in \mathbb{M}_{mm_a} : refines_a(refine_a(m_a), m_a) \qquad \text{❍}$$

Ist die Abstraktionsbeziehung zwischen zwei Arten von Modellen bekannt, so kann diese genutzt werden, um für eine gegebene Abbildung von abstrakteren auf konkretere Modelle nachzuweisen, dass es sich bei dieser Abbildung ggf. um eine korrekte Verfeinerungsabbildung bezüglich der gewählten Abstraktion handelt.

Eine mögliche Abstraktionsabbildung für hierarchisch organisierte Automaten kann beispielsweise so gewählt sein, dass durch die Abstraktionsabbildung sämtliche Subzustände eines Zustandes eliminiert werden. Bezüglich dieser Abstraktion ist jede Abbildung, die lediglich neue Subzustände und Transitionen zwischen diesen einführt und die ursprüngliche, oberste Hierarchieebene unverändert lässt, eine gültige Verfeinerungsabbildung.

Existiert zu einer Verfeinerungsabbildung eine Abstraktionsabbildung, so ist man weiterhin in der Lage zu überprüfen, ob ein Modell m_r das ggf. durch eine Verfeinerungsabbildung aus einem Modell m_a hervorgegangen ist und nachträglich manipuliert wurde, immer noch eine gültige Verfeinerung des Ursprungsmodells darstellt. Dies ist der Fall, falls gilt:

$$abstract(m_r) = m_a$$

Ein Sonderfall, in dem eine Verfeinerungsabbildung immer verfügbar ist, ist gegeben, falls es möglich ist zu einer Abstraktionsabbildung direkt eine Umkehrabbildung anzugeben. In diesem Sonderfall ist die Abstraktionsabbildung jedoch nicht nur surjektiv, sondern auch injektiv, d.h. sie ist bijektiv.

Die Möglichkeit festzustellen, ob ein gegebenes konzeptuelles Modell eine gültige Verfeinerung eines ursprünglichen Modells darstellt, ist in vielerlei Hinsicht hilfreich. So werden gerade die Ergebnisse der frühen Phasen eines Entwicklungsprojektes, wie z.B. die Anforderungen, im Projektverlauf oftmals nicht mehr kontinuierlich weiter gepflegt. Dies erschwert jedoch spätere Wartungsarbeiten erheblich und macht oftmals ein Reverse-Engineering bestehender Software notwendig, falls ein System nachträglich erweitert oder geändert werden soll [Sne98]. Durch die Überprüfung, ob eine Systemspezifikation noch eine gültige Verfeinerung seines abstrakten Modells der Anforderungen darstellt, lässt sich sicherstellen, dass die Anforderungsdokumentation durch den gesamten Projektverlauf hindurch konsistent ist.

Die Möglichkeit Inkonsistenzen zwischen Modellen verschiedener Abstraktionsebenen zu entdecken ist insbesonders bei sogenannten Multi-Stakeholder-Systemen relevant, bei denen Anwendungen unterschiedlicher Organisationen kooperieren sollen. Tritt ein Fehler bei solchen Anwendungen auf, so gilt es zunächst festzustellen, ob der Fehler durch technische Inkompatibilitäten, durch sich widersprechende Anforderungen der Partner oder durch eine fehlerhafte Umsetzung des Anforderungsmodelles eines der beteiligten Partner zustande kam. Eine detaillierte Diskussion dieser Problematik findet sich in [MS03a].

Ein für die Transformation konzeptueller Modelle geeigneter Mechanismus muss dementsprechend eine Reihe von Eigenschaften vorweisen. Im Folgenden sind die Anforderungen an eine Spezifikationssprache für die Transformation konzeptueller Modelle zusammengefasst:

Art der transformierten Modelle: Es müssen konzeptuelle Modelle durch metamodellkonforme Transformationen aufeinander abgebildet werden. Quell- und Zielmetamodelle müssen unterschiedlich sein dürfen.

Kardinalität der Ein-/Ausgabemodelle: Es werden 1:1-Transformationen benötigt, d.h. aus einem Quellmodell wird ein neues Zielmodell erzeugt.

Verwendetes Sprachparadigma: Deklarative Ansätze sind vorzuziehen, für primitive Typen können auch imperative Sprachkonstrukte eingesetzt werden.

Verarbeitung primitiver Typen: Primitive Typen müssen transformiert werden können. Es muss möglich sein, sowohl logische als auch arithmetische Operationen anzuwenden. Darüber hinaus müssen auch Zeichenketten verarbeitet werden können.

Bijektivität: Diese Eigenschaft ist ggf. wünschenswert um verifizieren zu können, dass ein Modell eine gültige Verfeinerung eines Quellmodells darstellt.

Rückverfolgbarkeit: Die Rückverfolgbarkeit der Entstehung von Elementen des Zielmodells ist wünschenswert.

Korrektheit von Spezifikationen: Es muss gewährleistet werden können, dass Spezifikationen ausführbar sind und ausschließlich Instanzen eines gewünschten Zielmetamodells erzeugen.

Fehlerverhalten: Die Sprache muss deterministisch sein und darf keine fehlerhaften Modelle erzeugen. Transformationsspezifikationen müssen bezüglich dieser Eigenschaft verifizierbar sein.

Mächtigkeit: Das Auffinden und Erzeugen komplexer Modellstrukturen im Quell- bzw. Zielmodell muss möglich sein.

Anwenderfreundlichkeit: Die Anwender sind Modellierungsexperten. Die Konzepte der Modellierungssprache sollten daher auch in der Sprache für Transformationsspezifikationen wiederverwendet werden. Spezifikationen sollten deklarativ und frei von undurchsichtigen Kontrollstrukturen sein. Die Kombinierbarkeit von Spezifikationen ist wünschenswert.

Ausführbarkeit: Eine automatisierte Ausführung ist erforderlich.

Effizienz: Die Effizient ist hier zweitrangig, da die automatisierte Verfeinerung konzeptueller Modelle eher selten erfolgt.

Realisierbarkeit: Eine Werkzeugunterstützung ist erforderlich.

Erweiterbarkeit: Zusätzliche Erweiterungen werden nicht benötigt.

2.2.4 Integration von Artefakten in ein konzeptuelles Modell

Um die Konsistenz der in unterschiedlichen Artefakten modellierten Informationen untereinander sicherzustellen, gilt es die Menge der in den Artefakten modellierten Informationen in ein gemeinsames konzeptuelles Modell zu integrieren. Die UML verfügt beispielsweise über neun verschiedene Diagrammtypen, deren Inhalt jeweils auf das konzeptuelle Modell der UML, eine Instanz des UML-Metamodells, abgebildet wird. Eine explizite Unterscheidung zwischen der abstrakten Syntax dieser Diagramme und dem UML-Metamodell existiert jedoch nicht. So werden innerhalb der UML Diagrammelemente in Form einer textuell spezifizierten eins-zu-eins Zuordnung auf Elemente des UML-Metamodells abgebildet. Beispielsweise lautet die Abbildungsvorschrift für Aktivitätssymbole aus UML-Aktivitätsdiagrammen in das UML-Modell folgendermaßen:

> „An action state symbol maps into an ActionState with the action-expression mapped to either the body of the entry action procedure of the State, or to a detailed action model within the procedure. The State is normally anonymous.“
> *(*[OMG02c]*, S. 3-161)*

Diese Art die Integration von Information aus Artefakten in ein gemeinsames konzeptuelles Modell zu spezifizieren ist jedoch mit einer Reihe gravierender Nachteile verbunden, die sich auch innerhalb der UML manifestieren. Da die abstrakte Syntax der UML-Diagrammtypen direkt auf Instanzen des UML-Metamodells abgebildet werden, finden sich im UML-Metamodell eine Vielzahl von Elementen deren Existenz oftmals lediglich durch die in den Diagrammen verwendete Notation begründet liegt. Dies führt zu der oftmals kritisierten Überfrachtung und Unübersichtlichkeit des UML-Metamodells [CEK+00]. Wünschenswert wäre an dieser Stelle ein möglichst kleines aber ausdrucksmächtiges Kernmodell zur Spezifikation von Systemen.

Ein weiterer hiermit verbundener Nachteil ist, dass eine Erweiterung der durch die UML unterstützten oder eine Veränderung der bestehenden Diagrammtypen zwangsläufig Änderungen im UML-Metamodell nach sich zieht. Zwar können UML-Diagramme durch die Verwendung der UML-eigenen Erweiterungsmechnismen an unterschiedliche Befürfnisse angepasst werden, in den meisten Fällen spiegelt ein so entstehendes Modell als Instanz des UML-Metamodells jedoch nicht mehr die durch die Erweiterungen beabsichtigte Semantik wieder. So wird beispielsweise in [HNS99] ein UML-Profil zur Modellierung von Software-Architekturen vorgestellt, zusätzlich aber ein eigenes Metamodell für die Modellierung solcher Architekturen angegeben.

Letztendlich verdeutlicht das oben angeführte Zitat auch den Mangel an Präzision, der mit natürlichsprachlichen Spezifikationen von Transformationen einhergeht. Diese Art der Spezifikation erlaubt es zwar direkte Abbildungen von Diagrammelementen in das UML-Modell zu beschreiben, komplexere Abbildungen zwischen der abstrakten Syntax einer Notation und einem konzeptuellen Modell sind auf diese Weise jedoch nicht hinreichend präzise formulierbar.

Um diese Mängel zu umgehen werden im hier vorgestellten Ansatz stattdessen die Modelle der abstrakten Syntax einer Notation in ein konzeptuelles Modell übersetzt. Hierzu wird für jede Notation eine Modelltransformationsspezifikation angegeben, die festlegt wie Elemente des abstrakten Syntaxmodells der Beschreibungstechnik in Sichten des konzeptuellen Modells abgebildet werden.

Definition 2.2.6 (Beschreibungstechnik)
Eine Beschreibungstechnik besteht aus

- einer Notation n und
- einer Modelltransformationsspezifikation mt, die festlegt, wie Instanzen der abstrakten Syntax der Notation auf Sichten eines konzeptuellen Modells mit dem Metamodell cmm abgebildet werden:

$$mt : \overline{\mathbb{M}}_{n|_{mm}} \rightarrow \overline{\mathbb{M}}_{cmm}$$

❍

Beschreibungstechniken stellen somit die Verbindung zwischen Prozessmodellen und konzeptuellen Modellen dar. Sie geben den im Prozessmodell eingeführten Notationen eine Semantik auf Basis eines konzeptuellen Modells, indem sie festlegen wie die Elemente der Notation auf Sichten eines konzeptuellen Modells abgebildet werden.

Artefakte gleichen Typs verwenden dieselbe Beschreibungstechnik. Es wird im Folgenden davon ausgegangen, dass durch eine Menge von Artefakten gleichen Typs ein gemeinsames Instanzmodell

der abstrakten Syntax ihrer jeweiligen Notation, spezifiziert wird. Diese Annahme impliziert, dass verschiedene Artefakte gleichen Typs keine Inkonsistenzen untereinander aufweisen dürfen. Ein Beispiel für eine solche Inkonsistenz ist eine Klasse, die in zwei verschiedenen UML Klassendiagrammen modelliert wird, wobei sich die Attribute der beiden Versionen der Klasse unterscheiden.

Da für die Erstellung von Artefakten in der Praxis Werkzeuge verwendet werden, kann generell davon ausgegangen werden, dass durch eine solche Werkzeugunterstützung bereits Inkonsistenzen im Modell der abstrakten Syntax einer Notation ausgeschlossen werden können. Für die Erstellung eines gemeinsamen Modells der abstrakten Syntax aus verschiedenen Artefakten können jedoch auch die im Rahmen dieses Abschnitts vorgestellten Techniken zur Integration verschiedener Sichten in ein konzeptuelles Modell verwendet werden.

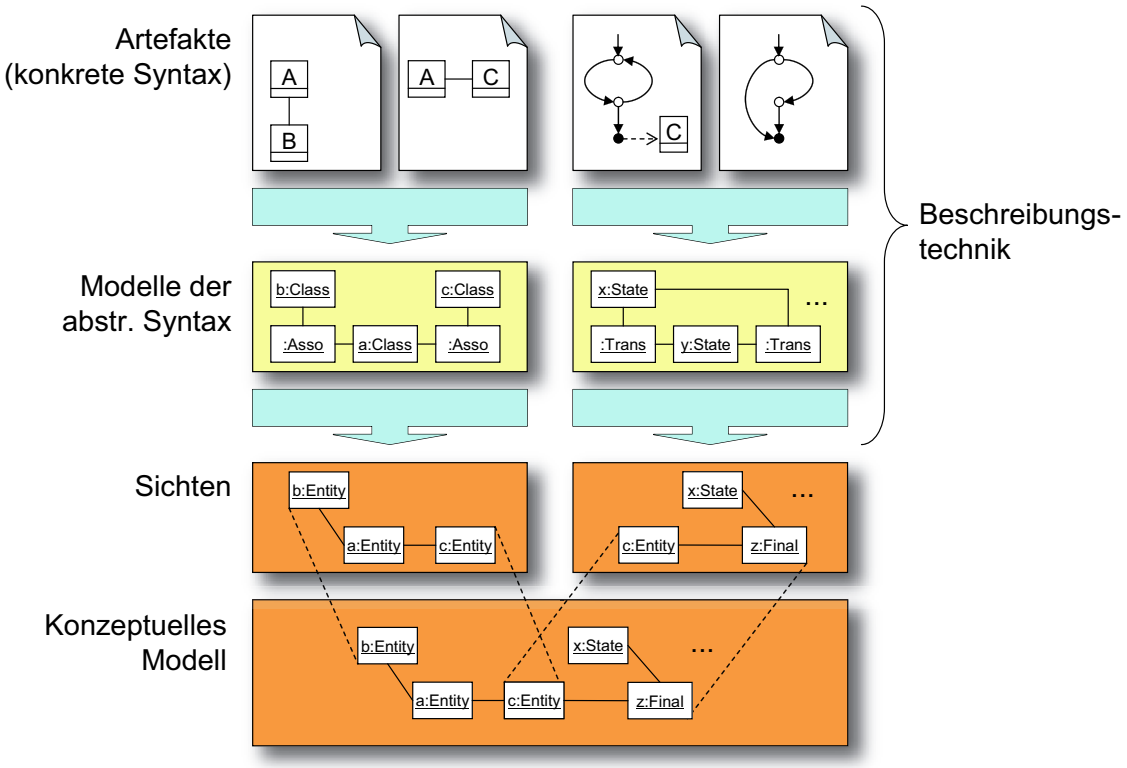


Abbildung 2.6: Integration von Artefakten in ein konzeptuelles Modell

Abbildung 2.6 skizziert den Sachverhalt exemplarisch. Im Beispiel werden vier Artefakte verschiedener Beschreibungstechniken in ein gemeinsames konzeptuelle Modell integriert. Die einzelnen Elemente des Diagramms werden im folgenden kurz erläutert:

Artefakte Artefakte liegen in einer graphischen oder textuellen Notation vor. Im Beispiel existieren jeweils zwei Artefakte des Typs Datenmodelldiagramm und Zustandsdiagramm. Für die Erstellung von Artefakten werden jeweils geeignete Modellierungswerkzeuge bzw. Editoren verwendet.

Modelle der abstrakten Syntax Die Elemente der konkreten Syntax der Notationen werden jeweils in ein gemeinsames Modell der abstrakten Syntax abgebildet (siehe Def. 2.1.9). In der Pra-

xis entspricht diese Abbildung meist dem Aufbau einer internen Datenstruktur innheralb eines Modellierungswerkzeugs. In der Abbildung sind die beiden Modelle der abstrakten Syntax in Form von Objektdiagrammen dargestellt. So besteht das Modell der abstrakten Syntax für Klassendigramme aus Objekten der Typen Class, Association, etc. Die Objekte des abstrakte Syntaxmodells der Automatendiagramme sind dagegen vom Typ State und Transition.

Sichten Gemäß der Modelltransformationsspezifikation der jeweilige Beschreibungstechnik werden die Modelle der abstrakten Syntax auf konzeptuelle Teilmodelle, bzw. Sichten des konzeptuellen Modells, abgebildet. Im Beispiel werden die Sichten wieder durch Objektmodelle repräsentiert. Alle Elemente der beiden Sichten sind Instanzen der Elemente des konzeptuellen Metamodells.

Konzeptuelles Modell Die einzelnen Sichten werden zu einem gemeinsamen konzeptuellen Modell zusammengefasst. Führt das Zusammenführen der verschiedenen Sichten zu einem nicht metamodellkonformen Modell oder ist ein Zusammenführen der Sichten nicht möglich, so ist die durch die verschiedenen Artefakte erstellte Systemspezifikation inkonsistent. Eine entsprechende Werkzeugunterstützung kann nun Art und Quelle des Fehlers an den Benutzer melden, der die Inkonsistenzen durch Überarbeitung der betroffenen Artefakte beheben kann.

Einzelne Sichten sind selbst wieder Modelle, die Aussagen über Aspekte des zu realisierenden Systems oder seiner Umwelt machen. Die Gesamtheit dieser Aussagen sollte sich in der Komposition dieser Sichten niederschlagen.

Definition 2.2.7 (Komposition)
Eine Komposition von Modellen bzw. Sichten mit dem gleichem Metamodell *mm* ist eine kommutative und assoziative Abbildung

$$\uplus : \overline{\mathbb{M}}_{mm} \times \cdots \times \overline{\mathbb{M}}_{mm} \rightarrow \overline{\mathbb{M}}_{mm} \cup \bot$$

Hierbei muss gelten, dass das Ergebnis einer Komposition genau die Informationen zur Spezifikation eines Systems enthält, die auch in den Teilmodellen/Sichten enthalten ist. Bildet die Gesamtheit der Teilmodelle/Sichten ein inkonsistentes Modell, so ist das Ergebnis der Komposition $\bot$. Für die Komposition wird die Infixschreibweise verwendet ($m_0 \uplus m_1$). ❍

Ein Kompositionsoperator für ein konzeptuelles Metamodell muss also so definiert werden, dass die semantische Interpretation des Ergebnismodells auch die Komposition der semantischen Interpretationen der Quellmodelle wiederspiegelt. Dementsprechend muss der Kompositionsoperator für jedes konzeptuelle Metamodell in Abhängigkeit vom zugrundeliegenden semantischen Modell eigens definiert werden. Bestehen Modelle beispielsweise aus einer Menge prädikatenlogischer Aussagen, so ist die Konjunktion in den allermeisten Fällen eine sinnvolle Interpretation des $\uplus$-Operators. Für objektorientierte Modelle wird in Abschnitt 3.5.3 (Def. 3.5.12, Seite 92) ein möglicher Kompositionsoperator angegeben, der für die meisten Anwendungen geeignet ist. Ob dieser jedoch für ein gegebene objektorientiertes Metamodell korrekt ist, hängt ausschließlich von der Semantik des Metamodells ab.

Im Gegensatz zur Transformation von Modellen gemäß Definition 2.2.1 existieren bei der Integration von Artefakten in ein gemeinsames konzeptuelles Modell mehrere Quellmodelle. Eine solche Abbildung mehrerer Modelle auf ein Modell wird im Folgenden als Integrationstransformation von Beschreibungstechniken bezeichnet.

Definition 2.2.8 (Integrationstransformation)
Eine Integrationstransformation für eine Menge von Modelltransformationen $mt_0, \ldots, mt_n$ mit dem gleichen Zielmetamodell mm und den Quellmetamodellen $mm_0, \ldots, mm_n$ ist eine Abbildung, mit

$$int : \overline{\mathbb{M}} \times \cdots \times \overline{\mathbb{M}} \rightarrow \overline{\mathbb{M}}_{mm}, \text{ mit}$$
$$int[mt_0, \ldots, mt_n](m_0, \ldots, m_n) := mt(m_0) \uplus \cdots \uplus mt(m_n) \quad \text{❍}$$

Dementsprechend ergibt sich für das Ergebnis der Integration einer Menge von Beschreibungstechniken $B = \{b_0, \ldots, b_n\}$ ein Modell m_{int} mit

$$\begin{aligned} m_{int} &= int[b_0|_{mt}, \ldots, b_n|_{mt}](m_0, \ldots, m_n) \\ &\overset{\text{Def. 2.2.8}}{=} b_0|_{mt}(m_0) \uplus \cdots \uplus b_n|_{mt}(m_n) \end{aligned}$$

Beschreibungstechniken bzw. die durch sie erzeugten Sichten sind *orthogonal*, falls sie keine Inkonsistenzen im konzeptuellen Modell erzeugen können. Diese Eigenschaft ist erfüllt, falls ihre Integrationstransformation in das konzeptuelle Modell immer metamodellkonform ist.

Definition 2.2.9 (Orthogonale Sichten/Beschreibungstechniken)
Eine Menge von Beschreibungstechniken $B = \{b_0, \ldots, b_n\}$ ist *orthogonal*, falls die ihre Integration in ein gemeinsames konzeptuelles Modell mit dem Metamodell mm_c niemals zu Inkonsistenzen führen kann, d.h. es gilt:

$$\forall m_0 \in \overline{\mathbb{M}}|_{b_0|n|_{mm}}, \ldots, m_n \in \overline{\mathbb{M}}|_{b_n|n|_{mm}} : int[b_0|_{mt}, \ldots, b_n|_{mt}](m0, \ldots, m_n) \in \overline{\mathbb{M}}_{mm_c} \quad \text{❍}$$

Ein Beispiel für zwei nicht-orthogonale Beschreibungstechniken der UML sind Sequenz- und Zustandsdiagramme. Offensichtlich behandeln beide Beschreibungstechniken Verhaltensaspekte von Systemen, ein Sequenzdiagramm kann somit ein exemplarische Verhalten spezifizieren, welches gemäß dem Zustandsdiagramm nicht Teil des Systemverhaltens sein kann.

Inkonsistenzen zwischen nicht orthogonalen Beschreibungstechniken bzw. Sichten können generell nicht ausgeschlossen werden. Es gilt jedoch, diese Inkonsistenzen im Verlauf der Integration von Artefakten in ein konzeptuelles Modell zu entdecken und den Entwickler damit zu konfrontieren.

Wie in Abschnitt 2.2.3 beschrieben gehen konzeptuelle Modelle auf verschiedenen Abstraktionsebenen durch Transformationen auseinander hervor. Ein Ziel dieses Ansatzes ist es, wie von der MDA vorgeschlagen, ein plattformspezifisches Modell vollkommen aus einem plattformunabhängigen Modell durch die Auswahl einer geeigneten Transformation erzeugen zu können. In der Praxis ist dies jedoch in den allermeisten Fällen nicht möglich. Verfeinert Modelle müssen in aller Regel um weitere Informationen angereichert werden. Dies geschieht durch die Integration neu erstellter Artefakte in das verfeinerte Modell. Abbildung 2.7 skizziert die Situation im Kontext einer MDA-basierten Entwicklung: einzelne Modelle werden schrittweise durch Transformationen verfeinert und durch zusätzliche Informationen aus Artefakten angereichert.

Wird nun davon ausgegangen, dass eine Modelltransformation eines Modells eine Verfeinerungabbildung bezüglich einer gewählten Abstraktion darstellt, so ist es von Interesse feststellen zu können, ob auch durch die zusätzliche Integration von Artefakten in das verfeinerte Modell die Verfeinerungsbeziehung zwischen dem so entstehenden Modell und dem ursprünglichen abstrakteren Modell erhalten bleibt. Hierzu wird der Begriff der *Verfeinerungskomposition* eingeführt.

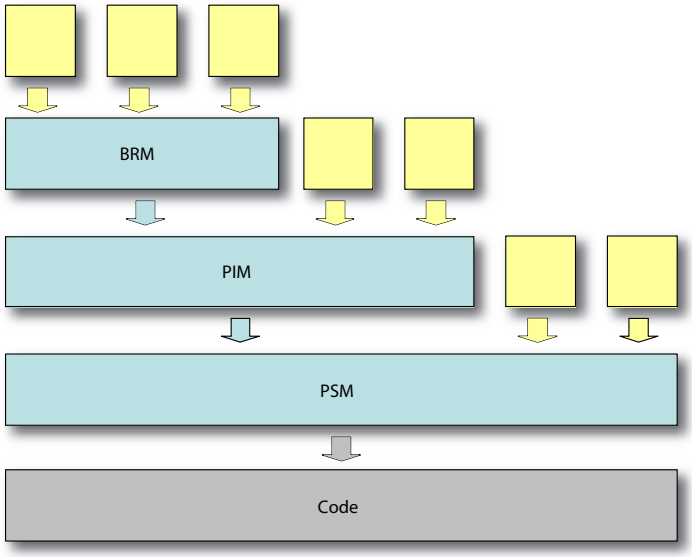


Abbildung 2.7: Verfeinerung von Modellen und Integration von Artefakten im Rahmen eines MDA-basierten Entwicklungsprozesses

Definition 2.2.10 (Verfeinerungskomposition)
Eine Integrationstransformation $int[mt_0,\ldots,mt_n](m_0,\ldots,m_n)$ ist eine *Verfeinerungskompostion* bezüglich einer Abstraktion $abstract_i$ des Modells $m_i \in \{m_0,\ldots,m_n\}$ falls gilt:

$$\forall\, m_0 \in \overline{\mathbb{M}}_{mm_0},\ldots,m_n \in \overline{\mathbb{M}}_{mm_n} : m_i = abstract_i\big(int[mt_0,\ldots,mt_n](m_0,\ldots,m_n)\big) \qquad \text{❍}$$

Bezeichnet $abstract_i$ die Abstraktionsabbildung zwischen zwei konzeptuellen Modellen, so ist sichergestellt, dass diese Abstraktionsbeziehung auch nach der Integration beliebiger Artefakte in das verfeinerte Modell erhalten bleibt, falls die Komposition der Integrationsabbildungen und der Verfeinerungsabbildung eine Verfeinerungskomposition bezüglich $abstract_i$ ist.

Nachstehend wir auf Basis der Definition 2.2.7 für den Kompositionsoperator eine Teilmodellrelation eingeführt:

Definition 2.2.11 (Teilmodell)
Ein Modell m_0 ist ein *Teilmodell* eines Modells m_1, falls die im folgenden definierte Infix-Relation $\trianglelefteq$ gilt:

$$\trianglelefteq : \overline{\mathbb{M}} \times \overline{\mathbb{M}} \to \mathbb{B}$$
$$m_0 \trianglelefteq m_1 :\Leftrightarrow \exists m \in \overline{\mathbb{M}} : m \uplus m_0 = m_1 \qquad \text{❍}$$

Gemäß Definition 2.2.11 enthält das Modell m_1 mindestens alle Aussagen die in m_0 über ein zu spezifizierendes System gemacht werden. Generell muss die $\trianglelefteq$ Relation auf Basis der $\uplus$-Operation für jede Art von Modellen eigens definiert werden. $m_0 \trianglelefteq m_1$ sollte jedoch nur dann gelten, falls das Modell m_1 mindestens denselben Informationsgehalt wie das Modell m_0 hat. Für objektorientierte Modelle wird in Abschnitt 3.1 eine mögliche Definition für eine Teilmodellrelation angegeben.

Im Idealfall ist die Verfeinerung eines konzeptuellen Modells und die Integration zusätzlicher Artefakte in das Ergebnis eine Verfeinerungskomposition. Dies ist dann der Fall, falls sämtliche zusätzliche Informationen aus der Integration der Artefakte durch die Abstraktionsabbildung wieder aus

dem so erweiterten Modell entfernt werden. Beispiele hierfür sind das Einführen von Subzuständen in Automaten oder Subkomponenten in Architekturbeschreibungen.

Als *konsistente Erweiterung* eines Modells bezüglich einer Abstraktion *abstract* wird eine Integrationstransformation angesehen, deren ursprüngliches Abstraktion immer ein *Teilmodell* des Ergebnisses der Integratinstransformation ist.

Definition 2.2.12 (Konsistente Erweiterung)
Eine Integrationstransformation $int[mt_0,\ldots,mt_n](m_0,\ldots,m_n)$ ist eine *konsistente Erweiterung* bezüglich einer Abstraktion $abstract_i$ des Modells $m_i \in \{m_0,\ldots,m_n\}$ falls gilt:

$$\forall m_0 \in \overline{\mathbb{M}}_{mm_0},\ldots,m_n \in \overline{\mathbb{M}}_{mm_n} : m_i \trianglelefteq abstract_i\big(int[mt_0,\ldots,mt_n](m_0,\ldots,m_n)\big) \qquad \text{❍}$$

In der Praxis sind konsistente Erweiterungen oftmals anzutreffen, wenn fachliche Aspekte, die in einem abstrakten Domänenmodell nicht berücksichtigt wurden, erst in einem verfeinertem technischem Modell Eingang finden. In einem solchen Fall ist es wichtig, die Konsistenz der Erweiterung mit dem bisherigen abstrakteren Modell verifizieren zu können und ggf. das abstrakte Modell mit der nun nicht mehr konsistenten Verfeinerung in Einklang zu bringen.

Durch die Abbildung von Artefakten in ein konzeptuelles Modell erhalten die Beschreibungstechniken implizit eine Semantik, da sich für jede aus einem Artefakt erzeugte Sicht ein semantisches Modell angeben lässt. Ein ähnlicher Ansatz findet sich in [Gog00]. Hier wird vorgeschlagen, die Elemente des UML-Metamodells, welche in erster Linie durch die UML-Notationen motiviert sind, auf ein Kernmodell des UML-Metamodells abzubilden. Eine formale Semantik muss dann lediglich für dieses Kernmetamodell angegeben werden. Die Semantik der übrigen Elemente ergibt sich durch ihre Abbildung auf das Kernmodell.

Um die Integration von Artefakten in ein konzeptuelles Modell zu ermöglichen, muss eine hierfür geeignete Modelltransformationssprache demnach die im Folgenden angeführten Anforderungen erfüllen:

Art der transformierten Modelle: Es müssen Modelle der abstrakten Syntax verschiedener Notationen durch metamodellkonforme Transformationen auf ein konzeptuelles Modell abgebildet werden.

Kardinalität der Ein-/Ausgabemodelle: Es werden n:1-Transformationen benötigt, d.h. mehrere Quellmodelle werden in ein gemeinsames Zielmodell transformiert.

Verwendetes Sprachparadigma: Deklarative Ansätze sind vorzuziehen, für primitive Typen können auch imperative Sprachkonstrukte eingesetzt werden.

Verarbeitung primitiver Typen: Primitive Typen müssen transformiert werden können. Es muss möglich sein, sowohl logische als auch arithmetische Operationen anzuwenden. Darüber hinaus müssen auch Zeichenketten verarbeitet werden können.

Bijektivität: Diese Eigenschaft ist wünschenswert um ggf. verifizieren zu können, ob durch verschiedene Artefakte nicht-orthogonale Aspekte modelliert wurden.

Rückverfolgbarkeit: Diese Eigenschaft ist wünschenswert, um bei Inkonsistenzen in einem konzeptuellen Modelle die Elemente in Artefakten identifizieren zu können, die für die Inkonsistenzen verantwortlich sind.

Korrektheit von Spezifikationen: Die Korrektheit von Spezifikationen zur Transformation von einzelnen Notationen muss gewährleistet sein. Die Konfliktfreiheit von Transformationsspezifikationen zur Integration verschiedener Artefakte sollte ggf. nachweisbar sein um orthogonale Beschreibungstechniken identifizieren zu können.

Fehlerverhalten: Fehler während der Integration von Artefakten zwischen ihnen sollten zum ergebnislosen Abbruch der Transformation und einen Hinweis auf die Art des Konflikts führen. Dies erlaubt es dem Benutzer, auf die Elemente in Artefakten hinzuweisen, durch die Inkonsistenzen verursacht wurden.

Mächtigkeit: Das Auffinden und Erzeugen von Modellstrukturen in den Quellmodellen bzw. dem Zielmodell muss möglich sein.

Anwenderfreundlichkeit: Die Anwender sind Modellierungsexperten. Die Konzepte der Modellierungssprache sollten daher auch in der Sprache für Transformationsspezifikationen wiederverwendet werden. Spezifikationen sollten deklarativ und frei von undurchsichtigen Kontrollstrukturen sein. Die Kombinierbarkeit von Spezifikationen ist erforderlich, um zusätzliche Artefakttypen berücksichtigen zu können.

Ausführbarkeit: Zumeist erfolgt die Integration von Artefakten in ein gemeinsames konzeptuelles Modell durch die Anbindung externer Modellierungswerkzeuge an ein gemeinsames Repository. Eine automatisierte Ausführung ist somit erforderlich.

Effizienz: Das Verfahren muss hinreichend Effizient sein, um beim Speichern eines Artefakts durch ein Modellierungswerkzeug die Modelle der abstrakten Syntax zur Laufzeit transformieren zu können.

Realisierbarkeit: Eine Werkzeugunterstützung ist erforderlich.

Erweiterbarkeit: Zusätzliche Erweiterungen der graphischen Syntax der Sprache können die Spezifikation von Transformationen bestimmter Notationen vereinfachen.

3 Die Bidirectional Object Oriented Transformation Language (BOTL)

Im Rahmen dieses Kapitels wird die „Bidirectional Object Oriented Transformation Language“ (BOTL) zur Transformation objektorientierter Modelle vorgestellt. Zunächst wird ein mathematisches Modell für objektorientierte Metamodelle und Modelle eingeführt, auf dessen Basis BOTL-Regeln und Regelwerke sowie eine graphische Notation für sie in Form eines UML-Profils, definiert werden. Im Anschluss wird die Semantik von BOTL-Regeln eingeführt, indem der Mechanismus zur Transformation von Modellen als mathematische Abbildung zwischen BOTL-Modellen definiert wird.

3.1 Objektorientierte Modelle und Metamodelle

Da sich in der Praxis objektorientierte Modelle, und im insbesonders die UML sowie die MOF, weitgehend etabliert haben, wird in dieser Arbeit davon ausgegangen, dass Modelle zur Beschreibung von Software-Systemen und Produktmodellen objektorientierte Modelle sind oder zumindest eine Repräsentation in Form eines objektorientierten Modells existiert, wie dies innerhalb der MOF vorgesehen ist.

Im Folgenden werden objektorientierte Modelle und Metamodelle als eine spezielle Form der in Kapitel 2.1 vorgestellten Modelle und Metamodelle eingeführt. Es wird zunächst eine formale Basis für objektorientierte Metamodelle, die üblicherweise Klassenmodelle genannt werden, und ihrer Instanzen, die als Objektmodelle bezeichnet werden, geschaffen. Da im weiteren Verlauf der Arbeit ausschließlich objektorientierte Modelle und Metamodelle betrachtet werden, werden diese zur einfacheren Lesbarkeit auch nur kurz als Modelle bzw. Metamodelle bezeichnet.

3.1.1 Primitive Typen und Identifikatoren

Zunächst erfolgt die Definition aller möglichen Identifikatoren. Diese werden benötigt, um Klassen, Objekte und primitive Datentypen unterscheiden zu können.

Definition 3.1.1 (Menge der Identifikatoren $\mathbb{ID}$)
Es sei $\mathbb{ID}$ die Menge aller gültigen *Identifikatoren*, wobei gelte: $|\mathbb{ID}| = \infty$. ❍

Ein primitiver Typ (oder kurz Typ) wird durch seinen eindeutigen Identifikator ausgewiesen. Er enthält eine (potentiell unendlich große) Menge von Werten.

Definition 3.1.2 (Typ)
Ein *Typ* ist ein Tupel (it, T_{val}), bestehend aus

- einem *Identifikator* $it \in \mathbb{ID}$ und
- einer Menge T_{val} von *Werten*.

$\mathbb{T}$ bezeichnet die Menge aller Typen. ❍

Typen in BOTL entsprechen prinzipiell den primitiven Datentypen in gängigen Programmiersprachen, wie z.B. Java. In Java sind typische primitive Datentypen beispielsweise int (nicht jedoch Integer) oder float. Oftmals kann es jedoch auch sinnvoll sein, komplexe Typen von Programmiersprachen (wie z.B. die Java-Klassen String oder Integer) im Kontext von BOTL-Transformationen ebenfalls wie primitive Typen zu behandeln.

Definition 3.1.3 (Typ-Belegung (*TA*))
Eine *Typbelegung TA* besteht aus einer Menge von Typen $\{t_0,\ldots,t_n\}$ für die gilt:

$$\forall t_i,t_j \in TA: \quad t_i|_{it} = t_j|_{it} \Rightarrow t_i = t_j$$ ❍

Demzufolge müssen alle Typen einer Typbelegung eindeutige Identifikatoren haben. Im Gegensatz zu gängigen Programmiersprachen sind die Mengen der möglichen Werte zweier Typen innerhalb des BOTL-Modells nicht notwendigerweise disjunkt. Während in BOTL ein Wert 42 zu den Typen int und long gehören kann, existieren in Java hierfür zwei unterschiedliche Werte 42 und 42L. Dies erlaubt es, im Verlauf einer Transformation Werte primitiven Typs direkt ineinander überzuführen ohne explizite Typkonversionen durchführen zu müssen.

3.1.2 Objektorientierte Metamodelle

Innerhalb dieses Abschnitts werden Klassenmodelle als Teil der Menge $\overline{\mathbb{MM}}$ aller Metamodelle formal definiert. Die Definition orientiert sich an den Standards MOF und UML, umfasst jedoch nicht alle dort auftretenden Konzepte. Für BOTL-Transformationen unerhebliche Eigenschaften wie z.B. Methoden oder abstrakte Klassen werden nicht berücksichtigt. Dafür wird zusätzlich das Konzept der Primärschlüssel eingeführt (Primary Keys). Die Werte einer Menge von Attributen, die als Primärschlüssel deklariert sind, bestimmen eineindeutig die Identität der Instanzen der jeweiligen Klasse.

Definition 3.1.4 (Klasse (*c*))
Die Menge aller denkbaren Klassen wird mit $\mathbb{C}$ bezeichnet. Eine *Klasse* $c \in \mathbb{C}$ ist ein Tupel $(id, Super, A, Keys)$, bestehend aus

- einem *Identifikator* $id \in \mathbb{ID}$,
- einer Menge von *Superklassen* $Super \subseteq \mathbb{C}$
- einer endlichen Menge A von *Attributen* (n,t), bestehend aus
  - einem *Identifikator* $n \in \mathbb{ID}$ und
  - einem *Typ* $t \in TA$

  mit den Eigenschaften:

$$A \supseteq Super|_A \wedge \tag{3.1}$$
$$\nexists (n_i,t_i),(n_j,t_j) \in A: \quad n_i = n_j \tag{3.2}$$

- einer Menge *Keys* aus Tupeln (n,t) mit $Keys \subseteq A$

Weiterhin sei *consistent* ein Relation, welche definiert, ob eine Klasse konsistent zu einer gegebenen Typbelegung ist[1]:

$$consistent(c, TA) :\Leftrightarrow \forall t \in c|_A|_t : t \in TA$$ ❍

Die Menge der Superklassen *Super* umfasst alle Klassen, von denen eine Klasse direkt oder indirekt erbt. (3.1) legt fest, dass die Menge der Attribute der Klasse auch alle geerbten Attribute beinhaltet. (3.2) stellt sicher, dass jedes Attribut der Klasse einen eindeutigen Namen innerhalb der Klasse hat.

Im weiteren Verlauf der Arbeit wird zu einer gegebenen Klasse $class \in \mathbb{C}$ oftmals die Menge $class \cup class|_{Super}$ benötigt. Beispielsweise können alle Objekte mit einem Assoziationsende verbunden sein, die von dem durch das Assoziationsende geforderten Typ oder aber einem Supertyp dieses Typs sind. Um komfortabel auf diese Menge zugreifen zu können, wird im Folgenden die Abbildung *types* eingeführt:

Definition 3.1.5 (*types*(*class*))
Sei *types* eine Abbildung, so dass gilt:

$$types : \mathbb{C} \rightarrow \mathscr{P}(\mathbb{C})$$
$$types(class) := \{class\} \cup class|_{Super}$$ ❍

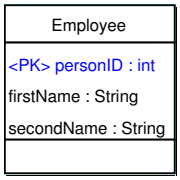


Abbildung 3.1: Beispiel für eine Klasse

Beispiel: Abbildung 3.1 stellt eine Klasse Employee mit drei Attributen personId und firstName und secondName dar. Das Attribut personId ist das einzige Primärschlüsselattribut der Klasse und als solches durch einen Tag <PK> gesondert gekennzeichnet. Zur Darstellung von BOTL-Klassen wird hier das BOTL-Profil für UML-Klassendiagramme verwendet, das in Abschnitt 3.4 näher vorgestellt wird. Für die graphische Darstellung von BOTL-Elementen wurde im Rahmen dieser Arbeit das in Abschnitt 6 vorgestellte Werkzeug zur Spezifikation von BOTL-Transformationen verwendet. Die dargestellte Klasse wird innerhalb des BOTL-Formalismus durch das Tupel

$$Emp := (Employee, \emptyset, \{(personId, int), (firstName, String), (secondName, String)\}, \{(personId, int)\})$$

repräsentiert. ❍

Eine Klassenbelegung bezeichnet eine Menge von Klassen, wie sie innerhalb eines Metamodells vorkommen können. Dementsprechend muss jede Klasse einer Klassenbelegung über einen eindeutigen Identifikator (d.h. Klassennamen) verfügen.

[1] Für eine Menge T von Tupeln $t = (x, y)$ sei $T|_x$ die Menge $\{t|_x : t \in T\}$.

Definition 3.1.6 (Klassenbelegung (*CB*))
Eine *Klassenbelegung CB* besteht aus einer endlichen Menge von Klassen $\{c_0, \ldots c_n\}$, für die gilt:

$$\forall c_i, c_j \in CB: \quad c_i|_{id} = c_j|_{id} \Rightarrow c_i = c_j \tag{3.3}$$

Zudem existiert eine irreflexive, antisymmetrische, transitive Vererbungsrelation $\rightarrowtriangle$, so dass für beliebige Klassen $c_0, c_1 \in CB$ gilt:

$$c_0 \rightarrowtriangle c_1 :\Leftrightarrow c_1 \in c_0|_{Super} \tag{3.4}$$

Die Relation *consistent* gilt, falls eine Klassenbelegung konsistent bezüglich einer gegebenen Typbelegung ist:

$$consistent(CB, TA) :\Leftrightarrow \forall c \in CB : consistent(c, TA)$$ ❍

Gemäß (3.3) müssen alle Klassen einer Klassenbelegung paarweise unterschiedliche Identifikatoren aufweisen. Die Relation $\rightarrowtriangle$ stellt in (3.4) sicher, dass keine Klasse einer Klassenbelegung von sich selbst erbt und keine zyklische Verbungsbeziehungen innerhalb einer Klassenbelegung existieren können. Die Relation $\rightarrowtriangle$ gilt hierbei auch für indirekte Vererbungsbeziehungen zwischen Klassen.

Klassen können mittels Klassenassoziationen zueinander in Bezug stehen. Im Gegensatz zu anderen Modellierungssprachen wie der UML beschränkt sich BOTL auch hier auf die für Transformationen notwendigen Kernkonzepte von Assoziationen. So sind innerhalb von BOTL lediglich binäre Assoziationen zulässig. Eine Abbildung von höherwertigen Assoziationen auf binäre ist jedoch ohne weiteres möglich und bei der Realisierung von Software-Systemen gängige Praxis.

Definition 3.1.7 (Klassenassoziation (*AE*))
Eine *Klassenassoziation AE* ist eine Menge von Tupeln der Form $ae = (rn, c, m, t, nav)$, so genannten *Klassenassoziationsenden*. Diese bestehen jeweils aus

- einem *Rollennamen* $rn \in \mathbb{ID}$,
- einer Klasse $c \in CB$,
- einer *Multiplizität* $m \in (\mathbb{N}_0 \cup \infty) \times (\mathbb{N} \cup \infty)$,
- einem *Aggregationstyp* $t \in \{\text{none}, \text{aggregate}, \text{composite}\}$ und
- einem Boolschen Wert zur Angabe der Navigierbarkeit *nav*

wobei gilt:

$$1 \leq |AE| \leq 2 \tag{3.5}$$
$$\wedge \ \left(|AE| = 2 \wedge \text{none} \in AE|_t \ \vee \ |AE| = 1 \wedge \text{none} = AE|_t\right) \tag{3.6}$$

Die Relation *consistent* gilt, wenn eine Klassenassoziation konsistent zu einer gegeben Klassenbelegung ist.

$$consistent(AE, CB) :\Leftrightarrow \forall c \in AE|_c : c \in CB$$

$\mathbb{AE}$ bezeichnet die Menge aller denkbaren Klassenassoziationen. ❍

3.5 legt fest, dass jede Klassenassoziation über entweder ein oder zwei Klassenassoziationsenden verfügen darf. Der Fall $|AE| = 1$ tritt genau dann auf, wenn eine Assoziation symmetrisch ist, wie in Abbildung 3.4 dargestellt. (3.6) stellt sicher, dass zumindest eines der Klassenassoziationsenden vom Aggregationstyp *none* ist, um wechselseitige Kompositionen oder Aggregationen einer Assoziation auszuschließen.

Eine Assoziation mit einem Assoziationsende vom Typ aggregate definiert eine asymmetrische, azyklische Relation zwischen den Instanzen der Klasse. Dementsprechend kann ein Objekt sich nicht direkt oder indirekt selbst aggregieren. Assoziationen mit einem Ende des Typs composite implizieren noch stärkere Einschränkungen für ihre Instanzen. Ein Objekt kann höchstens ein Container-Objekt haben zu dem es in einer composite Relation steht. Innerhalb von BOTL werden diese Aggregationstypen jedoch lediglich informell behandelt, d.h. die Korrektheit von Modelltransformationen bezüglich der Konsistenz von Aggregationen wird nicht formal überprüft. Somit ist der Aggregationstyp in erster Line für spätere Erweiterungen im Formalismus enthalten.

Auch die Navigierbarkeitseigenschaft *nav* hat im Rahmen des Formalismus lediglich informellen Charakter.

Abbildung 3.2: Die Klassenassoziation AE_1

Beispiel: Abbildung 3.2 zeigt eine Klassenassoziation zwischen den zwei Klassen Office und Employee. Multiplizitäten mit einem Wert von 1..1 werden im Diagramm nicht extra angegeben. Diese Klassenassoziation wird innerhalb des BOTL-Formalismus dargestellt durch die Menge

$$AE_1 = \{(worksIn, Office, (1,1), \text{none}, true), (employee, Employee, (0,2), \text{none}, true)\}$$

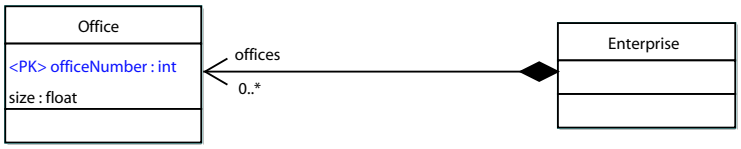


Abbildung 3.3: Die Klassenassoziation AE_2

Abbildung 3.3 zeigt eine in Pfeilrichtung unidirektional navigierbare Komposition, die aus der Menge von Klassenassoziationsenden

$$AE_2 = \{(offices, Office, (0,\infty), \text{none}, true), (\varepsilon, Enterprise, (1,1), \text{composite}, false)\}$$

besteht.

Eine symmetrische Klassenassoziation, wie sie beispielsweise zur Modellierung von Assoziationen der Art “Person ist_verwandt_mit Person” benötigt wird, ist in Abbildung 3.4 dargestellt. Innerhalb des BOTL-Formalismus wird diese Assoziation in Form der Menge

$$AE_3 = \{(inContactWith, Employee, (0,\infty), \text{none}, true)\}$$

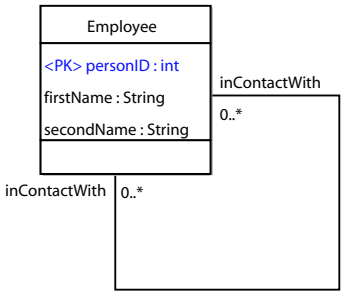


Abbildung 3.4: Die symmetrische Klassenassoziation AE_3

dargestellt. Sie besteht also lediglich aus einer *einelementigen* Menge von Assoziationsenden. ❍

Da im weiteren Verlauf oftmals auf das gegenüberliegende Ende eines Klassenassoziationsendes zugegriffen werden muss, wird hierfür die Funktion $oppositeEnd$ eingeführt:

Definition 3.1.8 (*oppositeEnd*(AE, ae))
Die Funktion *oppositeEnd* liefert das gegenüberliegende Ende eines Assoziationsendes *ae* innerhalb einer Klassenassoziation *AE*.

$$oppositeEnd(AE, ae) = \begin{cases} ae & \text{falls } AE = \{ae\} \\ ae^* & \text{falls } |AE| = 2 \text{ mit } AE = \{ae, ae^*\} \\ \bot & \text{sonst} \end{cases}$$ ❍

Ein Metamodell setzt sich aus einer Menge von Klassen zusammen, die über Klassenassoziationen miteinander in Bezug stehen. Es entspricht im wesentlichen UML- oder MOF-Klassenmodellen.

Definition 3.1.9 (Objektorientiertes Metamodell (*mm*))
Ein *objektorientiertes Metamodell mm* ist ein Tupel $(mmid, TA, CB, CA)$, bestehend aus

- einem eindeutigen Identifikator $mmid \in \mathbb{ID}$,
- einer Typbelegung *TA*,
- einer Klassenbelegung *CB* mit $consistent(CB, TA)$ und
- einer endlichen Menge von Klassenassoziationen *CA* mit:

$$\forall AE \in CA : consistent(AE, CB) \tag{3.7}$$

Die Menge aller objektorientierten Metamodelle wird mit $\mathbb{MM} \subset \overline{\mathbb{MM}}$ bezeichnet. ❍

(3.7) stellt sicher, dass die Enden aller Assoziationen ausschließlich mit Klassen aus der Klassenbelegung $mm|_{CB}$ des Metamodells verbunden sind.

Durch den Identifikator *mmid* wird sichergestellt, dass verschiedene Metamodelle zwar die gleichen Typen, Klassen und Assoziationen enthalten können, aber dennoch unterscheidbar sind. Diese Möglichkeit wird beispielsweise benötigt, falls verschiedenen Modelle strukturell gleiche Metamodelle haben, die jedoch unterschiedlich interpretiert werden, z.B. bei der Verwendung derselben Beschreibungstechnik mit jeweils unterschiedlicher Semantik. Für diesen Zweck wird die Relation $\cong$ für isomorphe Metamodelle eingeführt.

Definition 3.1.10 (Isomorphe Metamodelle)
Zwei Metamodelle $mm_0 \in \mathbb{MM}$ und $mm_1 \in \mathbb{MM}$ heißen genau dann *isomorph*, falls die im Folgenden definierte Relation $mm_0 \cong mm_1$ gilt:

$$\begin{aligned} mm_0 \cong mm_1 \Leftrightarrow : & mm_1|_{TA} = mm_0|_{TA} \wedge \\ & mm_1|_{CB} = mm_0|_{CB} \wedge \\ & mm_1|_{CA} = mm_0|_{CA} \wedge \end{aligned}$$ ❍

Da sich die Konzepte von Modellen und Metamodellen für die Software-Entwicklung, wie z.B. *Klasse* und *Assoziation*, oftmals mit den innerhalb des BOTL-Formalismus verwendeten überschneiden, wird im folgenden ein Beispiel aus einer anderen Anwendungsdomäne eingeführt. Das Beispiel wurde bereits in [BM03a] und [BM03b] zur Veranschaulichung von BOTL-Transformationen herangezogen. Zunächst werden zwei einfache und intuitive Metamodelle vorgestellt, welche die Struktur von Modellen zur Verwaltung von Personen, Büros und Telefonen innerhalb einer Organisation spezifizieren. Im weiteren Verlauf der Arbeit dienen diese Metamodelle als Grundlage für die Spezifikation einer Modelltransformation zwischen ihren Instanzen.

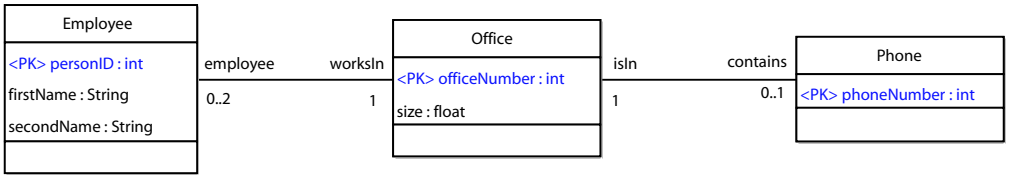


Abbildung 3.5: Das Metamodell mm_α

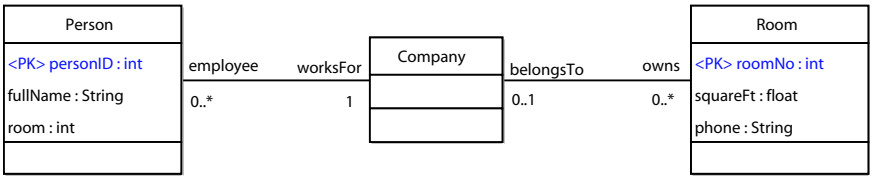


Abbildung 3.6: Das Metamodell mm_β

Beispiel: Abbildung 3.5 und Abbildung 3.6 zeigen zwei Beispiele für Metamodelle. Das Metamodell mm_α legt fest, dass Instanzen des Typs Office mit maximal zwei Employee-Instanzen und optional einer Phone-Instanz assoziiert sein dürfen. Die Attribute officeNumber, personID und *mm*phoneNumber dienen jeweils als Primärschlüssel für ihre Klassen.

Durch Instanzen des Metamodells mm_β können prinzipiell dieselben Informationen modelliert werden. Jedoch sind hier Objekte des Typs Person und Room direkt mit einem Company-Objekt verbunden. Die Telefonnummer (Attribut phone) ist in diesem Metamodell ein Attribut der Klasse Room. Die Zuordnung von Personen zu Räumen geschieht hier durch die Angaben der Raumnummer im Attribut room eines Objektes vom Typ Person. Auch hier verfügen die Klassen Room und Person über Primärschlüssel zur Bestimmung der Identität ihrer Instanzen. ❍

3.1.3 Objektorientierte Modelle

Objektorientierte Modelle sind die Instanzen objektorientierter Metamodelle. Im Rahmen dieses Abschnitts wird eine formale Definition objektorientierter Modelle vorgestellt und festgelegt, wann ein solches Modell konform zu einem Metamodell ist.

Instanzen von Klassen werden Objekte genannt. Innerhalb von BOTL muss zunächst ein Metamodell definiert werden, bevor Objekte instanziert werden können. Jedes Objekt hat einen für für seine Klasse eindeutigen Identifikator, d.h. es existieren keine zwei Objekte derselben Klasse mit dem gleichen Identifikator. Sind für eine Klasse Attribute als Primärschlüssel definiert, so legen die Attributwerte in den Objekten dieser Klasse deren Identität fest.

Definition 3.1.11 (Objekt (o))
Ein *Objekt o* ist ein Tupel (oi, ot, V), bestehend aus

- einem Identifikator $oi \in \mathbb{ID}$,
- einem *Objekttyp* $ot \in mm|_{CB}$ und
- einer Menge V von Attributen, bestehend aus Tupeln $(a,v) \in \mathbb{ID} \times \mathbb{T}|_{T_{\text{val}}} \cup \{\diamondsuit\}$ mit:

$$ot|_A|_n = V|_a \tag{3.8}$$

Die Relation *consistent* gilt, falls ein Objekt konsistent zu einem gegebenen Metamodell ist.

$$consistent(o, mm) :\Leftrightarrow o|_{ot} \in mm|_{CB} \wedge \tag{3.9}$$
$$\forall (a,v) \in o|_V : \exists t : \left((a,t) \in ot|_A \wedge v \in t|_{T_{Val}} \cup \{\diamondsuit\}\right) \tag{3.10}$$

Für Objekte, die Instanzen von Klassen mit Primärschlüsseln sind, muss weiterhin gelten:
$o|_{ot}|_{Keys} \neq \emptyset \Rightarrow$ Es existiert genau *eine* injektive Abbildung pK für alle Objekte des Typs $o|_{ot}$, die für jedes Objekt den Identifikator $o|_{oi}$ aus den Werten der Primärschlüsselattribute berechnet:

$$pK : \{(a,v) \in o|_V : \exists (n,t) \in o|_{ot}|_{Keys} \text{ mit } a = n\} \rightarrow \mathbb{ID} \tag{3.11}$$

Um einfach auf die Attributwerte von Objekten zugreifen zu können, wird die folgende Notation eingeführt:

$$o.att := \begin{cases} v & \text{falls } \exists (a,v) \in o|_V : a = att \\ \perp & \text{sonst} \end{cases}$$

Die Menge aller Objekte wird mit $\mathbb{O}$ bezeichnet. Ein Objekt o mit: $\forall att \in o|_V|_a : o.att \neq \diamondsuit$ heisst *$\diamondsuit$-frei*. ❍

(3.8) verwendet den projektiven |-Operator auf Mengen, um sicherzustellen, dass jedes Objekt exakt über die in der Klasse spezifizierten Attribute verfügt.

Um die Konsistenz eines Objektes bezüglich eines Metamodells zu gewährleisten, legt (3.9) fest, dass der Typ eines Objekts im gegebenen Metamodell vorkommt, während (3.10) sicherstellt, dass die Attributwerte des Objekts jeweils von dem in der Klasse definierten Typ sind oder den Wert $\diamondsuit$ aufweisen.

(3.11) legt schließlich fest, dass die Identität eines Objektes mit Primärschlüsselattributen von deren jeweiligen Attributwerten abhängt. Hierzu wird gefordert, dass eine injektive Abbildung existiert,

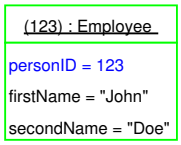


Abbildung 3.7: Ein Objekt

die es erlaubt den Objektidentifikator aus den entsprechenden Attributwerten zu berechnen. Da diese Abbildung in der Praxis zumeist nicht relevant ist, kann angenommen werden, dass sie als eindeutigen Objektidentifikator ein Tupel aus den Werten der Primärschlüsselattribute liefert.

Beispiel: Abbildung 3.7 zeigt ein Objekt der Klasse Employee mit dem Identifikator (123), drei Attributen personID, firstName, secondName und ihren Werten. Der Typ dieses Objekts bzw. die entsprechende Klasse ist in Abbildung 3.1 (siehe S. 53) dargestellt. Der Identifikator wurde anhand des Primärschlüsselattributs personID berechnet. Innerhalb des BOTL-Formalismus wird das Objekt durch das Tupel

$$((123), Employee, \{(personID, 123), (firstName, "John"), (secondName, "Doe")\})$$

dargestellt. Der Ausdruck (123).firstName liefert für dieses Objekt den Wert "John". ❍

Analog zur Klassenbelegung wird nun der Begriff *Objektbelegung* definiert. Eine Objektbelegung ist eine Menge von Objekten, wie sie innerhalb eines Modells auftreten kann, d.h. jedes Objekt verfügt über einen Klassen-weit eindeutigen Identifikator.

Definition 3.1.12 (Objektbelegung (*OB*))
Eine *Objektbelegung OB* besteht aus einer endlichen Menge von Objekten $\{o_0, \ldots, o_n\}$, für die gilt:

$$\forall o_i, o_j \in OB: \quad o_i|_{oi} = o_j|_{oi} \Rightarrow o_i = o_j \vee o_i|_{ot} \neq o_j|_{ot} \tag{3.12}$$

$\mathbb{OB}$ bezeichnet die Menge aller möglichen Objektbelegungen. Die Relation *consistent* gilt, falls eine Objektbelegung konsistent zu einem gegebenen Metamodell ist:

$$consistent(OB, mm) :\Leftrightarrow \forall o \in OB: consistent(o, mm)$$

$\mathbb{OB}_{mm}$ bezeichnet die Menge aller möglichen Objektbelegungen die konsistent zu einem Metamodell $mm \in \mathbb{MM}$ sind. Eine Objektbelegung *OB* heißt $\diamondsuit$*-frei*, falls alle Objekte $o \in OB$ $\diamondsuit$-frei sind. ❍

(3.12) legt fest, dass alle Objekt gleichen Typs in einer Objektbelegung unterschiedliche Identifikatoren haben. Ist eine Objektbelegung zudem konsistent zu einem Metamodell, so müssen sämtliche Objekte der Objektbelegung konsistent zu diesem Metamodell sein.

Objekte können über Objektassoziationen miteinander in Bezug stehen. So wie Objekte Instanzen von Klassen sind, sind Objektassoziationen Instanzen von Klassenassoziationen[2].

Definition 3.1.13 (Objektassoziation (*oa*))
Eine *Objektassoziation oa* bezüglich einer Objektbelegung $OB \in \mathbb{OB}$ ist ein Tupel $(OAT, card, OAE)$, bestehend aus

[2]Innerhalb der OMG wird für Objektassoziationen zumeist der Begriff „Link“ verwendet.

- einer Klassenassoziation $OAT \in mm|_{CA}$,
- einer *Kardinalität* $card \in \mathbb{N}^+$ und
- einer Menge OAE von *Objektassoziationsenden* $oae = (ae, o)$, bestehend aus
  - einem *Klassenassoziationsende* ae und
  - einem Objekt $o \in OB$

  so dass gilt:

$$OAE|_{ae} = OAT \quad \wedge \tag{3.13}$$
$$\forall oae \in OAE : oae|_{ae}|_c \in types(oae|_o|_{ot}) \tag{3.14}$$

Die Relation *consistent* gibt an, ob eine Objektassoziation konsistent zu einer Objektbelegung ist.

$$consistent(oa, OB) :\Leftrightarrow \forall o \in oa|_{OAE}|_o : o \in OB$$

Die Menge aller Objektassoziationen wird mit $\mathbb{OA}$ bezeichnet. ❍

Die Enden von Objektassoziationen haben dieselben Klassenassoziationsenden $OAE|_{ae}$ wie die Klassenassoziation OAT, von denen die Objektassoziation eine Instanz ist (3.13). Aussage (3.14) stellt sicher, dass der durch das das Klassenassoziationsende festgelegte Typ entweder vom Typ des Objektes oder einer Superklasse des Objekttyps ist.

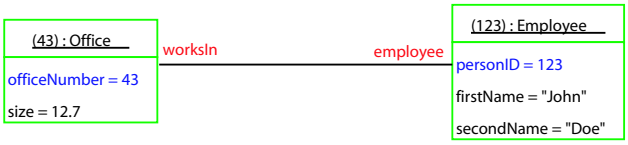


Abbildung 3.8: Eine Objektassoziation

Beispiel: In Abbildung 3.8 ist eine Objektassoziation als Instanz der Klassenassoziation aus Abbildung 3.2 dargestellt. Im BOTL-Formalismus wird sie durch das Tupel

$$(\, AE_1, 1, \{\, ((worksIn, Office, (1,1), \text{none}, true), 43),$$
$$((employee, Employee, (0,2), \text{none}, true), 123)\, \}\,)$$

repräsentiert.

$|OAE| = 1$ gilt für symmetrische Objektassoziationen. Abbildung 3.9 stellt exemplarisch eine symmetrische Objektassoziation mit der Kardinalität 2 dar. Diese Objektassoziation ist eine Instanz der Klassenassoziation aus Abbildung 3.4. Formal wird die Objektassoziation durch das Tupel

$$(AE_3, 2, \{((inContactWith, Employee, (0, \infty), \text{none}, true), 123)\})$$

dargestellt. ❍

Innerhalb der UML werden zwei gleichartige und gleichgerichtete Links zwischen denselben beiden

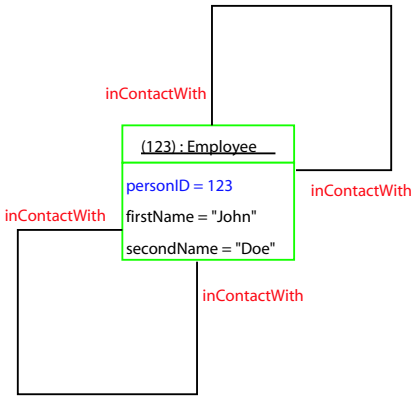


Abbildung 3.9: Eine symmetrische Objektassoziation

Objekten als zwei eigenständige Link-Instanzen angesehen. Im Gegensatz dazu werden diese innerhalb von BOTL als eine einzige Assoziation mit der Kardinalität 2 behandelt. Objektassoziationen verfügen also über keinen eindeutigen Identifikator und somit auch keine Identität. Dementsprechend ist es nicht möglich zwei Objektassoziationen desselben Typs zwischen denselben zwei Objekten zu unterscheiden.

Im Folgenden werden *objektorientierte Modelle* formal definiert. Ein Modell besteht aus Instanzen der Elemente eines Metamodells, also Objekten und Objektassoziationen. Obwohl für diese Elemente bereits eine Reihe von Konsistenzen gefordert werden, garantieren diese noch nicht, dass eine konsistente Objektbelegung zusammen mit einer Menge von konsistenten Objektassoziationen konform einem gegebenen Metamodell ist. So können ggf. die Kardinalitäten der Objektassoziation den zulässigen Höchstwert der Multiplizität der Klassenassoziation überschreiten bzw. den geforderten Mindestwert unterschreiten. Diese Eigenschaft wird im Folgenden für Modelle gefordert.

Definition 3.1.14 (Objektorientiertes Modell (m))
Ein *objektorientiertes Modell* m ist ein Tupel (mm, OB, OA) bestehend aus

- einem Metamodell $mm \in \mathbb{MM}$,
- einer $\Diamond$-freien Objektbelegung $OB \in \mathbb{OB}$ mit $consistent(OB, mm)$ und
- einer endlichen Menge von Objektassoziationen OA mit

  $$\forall oa \in OA : consistent(oa, OB)$$

  wobei gilt:

$$\forall AE \in mm|_{CA}, ae \in AE \text{ mit } (lb, ub) := oppositeEnd(AE, ae)|_m,$$
$$o \in OB \text{ mit } ae|_c \in types(o|_{ot}) :$$
$$lb \leq \sum_{\substack{oa \in OA : oa|_{oat} = AE \ \wedge \\ (ae, o) \in oa|_{OAE}}} oa|_{card} \leq ub \tag{3.15}$$
$$\forall oa \in OA, oae \in oa|_{OAE} :$$
$$oae|_o \in OB \wedge oae|_{ot} \in mm|_{CA} \tag{3.16}$$

Die Menge aller objektorientierter Modelle wird mit $\mathbb{M} \subset \overline{\mathbb{M}}$ bezeichnet.
Sind diese Eigenschaften eines Modells *m* für ein Metamodell *mm* erfüllt, so sagt man: Das Modell *m* ist *konform* zum Metamodell *mm*. ❍

Da im weiteren Verlauf der Arbeit in erster Linie objektorientierte Modelle behandelt werden, wird anstelle des Begriffs objektorientierter (Meta-)Modelle auch oftmals verkürzt von (Meta-)Modellen gesprochen.

(3.15) stellt sicher, dass in einem Modell für jedes Objekt die im Metamodell jeweils geforderte Anzahl von ausgehenden Objektassoziationen eines Typs vorhanden ist. Hierzu wird die Summe über alle von einem Objekt ausgehenden Objektassoziationen des jeweiligen Typs gebildet.

Das Metamodell aus Abbildung 3.4 (siehe S. 56) fordert beispielsweise, dass alle Objekte der Klasse A mindestens zwei, aber nicht mehr als acht, ausgehende Assoziationen eines bestimmten Typs haben müssen. Sollte dieses Konsistenzkriterium für mindestens ein Objekt nicht erfüllt sein, so wird Forderung (3.15) verletzt und das entsprechende Objektgeflecht stellt kein Modell bezüglich des gewählten Metamodells dar.

(3.16) stellt sicher, dass alle Objektassoziationen Objekte des Modells verbinden. Dies impliziert, dass keine freihängenden Assoziationen in einem Modell existieren dürfen. Die Eigenschaft, dass Objektassoziationen nur Objekte passenden Typs verbinden dürfen, wird bereits in der Definition von Objektassoziationen gefordert (siehe Def. 3.1.13 (3.14)) und muss deshalb an dieser Stelle nicht erneut verlangt werden.

Gemäß der Definition ist also *jedes* Modell konform zu seinem Metamodell. In Abschnitt 3.5 wird der Begriff *Modellfragment* für Objektstrukturen die nicht konform zu einem Metamodell sind eingeführt. Die nachstehende Definition führt eine Kurzschreibweise für die Menge aller Modelle, die konform zu einem Metamodell sind, ein:

Definition 3.1.15 (Menge konformer Modelle ($\mathbb{M}_{mm}$))
$\mathbb{M}_{mm_i}$ bezeichnet die Menge aller gültigen Modelle, die konform zu dem objektorientierten Metamodell mm_i sind, d.h.

$$\mathbb{M}_{mm_i} := \{m \in \mathbb{M} : m_{mm} = mm_i\}$$ ❍

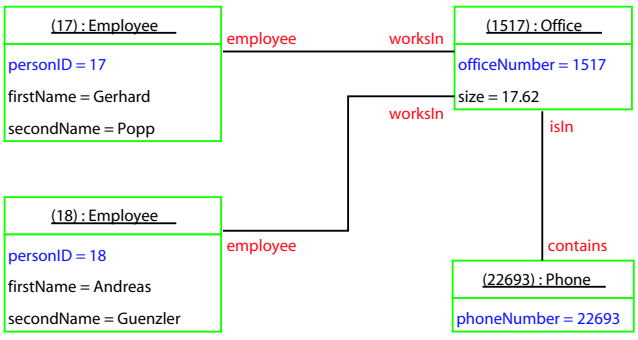


Abbildung 3.10: Das objektorientierte Modell m_α

Beispiel: Abbildung 3.10 zeigt das Modell $m_\alpha \in \mathbb{M}_{mm_\alpha}$. Es handelt sich hierbei um eine gültige Instanz des in Abschnitt 3.1.2 (S. 57) vorgestellten Metamodells mm_α. Offensichtlich sind sämtliche Multiplizitätsober- und untergrenzen, die im Metamodell mm_α gefordert sind, eingehalten. ❍

Generell existieren nicht zu jedem objektorientierten Metamodell konforme Modelle, die weder leer noch unendlich groß sind. Der folgende Abschnitt diskutiert diesen Sachverhalt und führt ein Verfahren ein, mit dem sich nachweisen lässt, dass zu einem Metamodell ggf. nicht-leere, endliche, konforme Modelle existieren.

3.2 Instanzierbarkeit von Metamodellen

Bei der Erstellung konzeptueller Metamodelle ist es oftmals wichtig sicherstellen zu können, dass es möglich ist, konforme Instanzen zu diesen Modellen zu erzeugen. Bei der implementierungsnahen Modellierung von Software-Systemen tritt dieses Problem oftmals nicht direkt zu Tage, da hier zumeist 1-zu-$*$- oder 0-zu-$*$-Assoziationen verwendet werden. Sollen in konzeptuellen Modellen allerdings auch Beschränkungen des Anwendungsumfeldes oder der eingesetzten technischen Plattformen abgebildet werden, so ist der Nachweis, dass ein solches konzeptuelles Modell grundsätzlich instanzierbar ist, unerlässlich.

Mögliche Constraints die der Instanzierbarkeit von Metamodellen im Wege stehen können sind in der Regel Aussagen der Art „Ein Kanal verbindet genau zwei Ports.“, „Der Server bedient maximal 32 Clients.“ oder „Ein Kunde muss mindestens eine und höchstens drei Lieferadressen angeben.“. Die nachfolgende Definition legt fest, dass ein Metamodell nur dann als *instanzierbar* bezeichnet wird, falls es möglich ist, ein zu dem Metamodell konformes Modell zu bilden, welches nicht leer und endlich ist.

Definition 3.2.1 (Instanzierbares Metamodell)
Ein Metamodell *mm* heißt genau dann *instanzierbar* (kurz $instancable(mm)$), falls das im Folgenden definierte Prädikat gilt:

$$instancable(mm) :\Leftrightarrow \exists m \in \mathbb{M}_{mm} : 0 \neq \left|m|_{OB}\right| \neq \infty \qquad \text{❍}$$

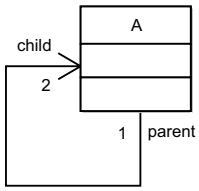


Abbildung 3.11: Minimalbeispiel für ein *nicht* instanzierbares Metamodell

Die Eigenschaft, dass ein Metamodell instanzierbar ist, ist keineswegs immer erfüllt. Abbildung 3.11 stellt ein Minimalbeispiel für ein nicht instanzierbares Metamodell dar. Jede Instanz der Klasse *A* muss genau zwei ausgehende Assoziationen zu „Kindern“ haben, darf aber selbst nur eine eingehende Assoziation dieses Typs haben. Jeder Versuch, eine zu dem Metamodell aus Abbildung 3.11 konformes Modell anzugeben, führt entweder zu einem leeren oder aber zu einem unendlich großen Modell.

Im Folgenden wird der Begriff *Konfiguration einer Klasse* eingeführt. Eine Konfiguration einer Klasse charakterisiert für ein gegebenes Metamodell die Menge aller Instanzen mit den gleichen Kardinaltiäten von ein- und ausgehenden Assoziationsenden. Für das Beispiel aus Abbildung 3.11 existiert demnach nur eine möglich Konfiguration der Klasse *A*. Diese ist mit genau zwei „ausgehenden“ Assoziationsenden des Typs parent und einem „eingehenden“ Assoziationsende des Typs child verbunden.

Definition 3.2.2 (Konfigurationen einer Klasse)
Eine *Konfiguration* CF^i_{class} einer Klasse $class \in \mathbb{C}$ innerhalb eines Metamodells $mm \in \mathbb{MM}$ ist eine Menge aus Tupeln $cf := (ASSO, end, card)$, bestehend aus:

- einer Klassenassoziation *ASSO* des Metamodells, d.h.

  $$ASSO \in mm|_{CA}$$

- einem Klassenassoziationsende *end*, welches mit der Klasse oder einer ihrer Superklassen verbunden ist, d.h.

  $$end \in ASSO \wedge end|_c \in types(class)$$

- der Anzahl $card \in \mathbb{N}^+$, der in dieser Konfiguration mit den Instanzen verbundenen Assoziationsenden des Typs *end*, d.h. für $(lb, ub) := oppositeEnd(ASSO, end)|_m$ muss gelten:

  $$lb \leq card \leq ub$$

wobei gilt:

$$\forall AE \in mm|_{CA}, ae \in AE \text{ mit } ae|_c \in types(class):$$
$$\exists_1 (A, e, n) \in CF^i_{class} : A = AE \wedge e = ae$$

Die Menge aller möglichen Konfigurationen einer Klasse *class* wird mit $\mathbb{CF}_{class}$ bezeichnet. Die Abbildung $configs(mm, class)$ liefert für ein Metamodell *mm* die Menge aller Konfigurationen der Klasse *class*. Zudem sei die Funktion *getCard* in folgender Weise definiert:

$$getCard(CF, ASSO, end) = \begin{cases} c & \text{falls } \exists (a, e, c) \in CF \text{ mit } a = ASSO, e = end \\ 0 & \text{sonst} \end{cases}$$ ❍

Eine Konfiguration einer Klasse bezeichnet somit die Menge ihrer Instanzen in beliebigen Modellen, die von jedem ausgehenden Assoziationstyp genau die gleiche Anzahl von Assoziationen hat.

Weiterhin liefert $getCard(CF, ASSO, end)$ die Kardinalität einer mit dem Assoziationsende *end* ausgehenden Assoziation *ASSO* in einer Konfiguration *CF*. Existiert kein solcher Assoziationstyp für die entsprechende Klasse so wird 0 zurückgeliefert.

Beispiel: Abbildung 3.12 zeigt eine leicht abgewandelte Variante des Metamodells aus Abbildung 3.11. Jedes Objekt vom Typ *A* kann nun entweder eine oder zwei ausgehende *child*-Assoziationen und eine oder keine ausgehende *parent*-Assoziation haben. Um Bezug auf diese einzige Assoziation des Metamodells nehmen zu können, wird diese im Weiteren als

$$PC := \{(parent, A, (0,1), \text{none}, false), (child, A, (1,2), \text{none}, true)\}$$

bezeichnet.

Gemäß Definition 3.2.2 liefert die Abbildung $configs(mm, A)$ die Menge aller Konfigurationen der Klasse *A* im Metamodell *mm*:

$$\begin{aligned} configs(mm, A) = \{ \quad & \{(PC, parent, 1), (PC, child, 0)\}, \\ & \{(PC, parent, 2), (PC, child, 0)\}, \\ & \{(PC, parent, 1), (PC, child, 1)\}, \\ & \{(PC, parent, 2), (PC, child, 2)\} \ \} \end{aligned}$$

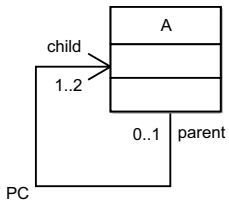


Abbildung 3.12: Das Metamodell *mm*

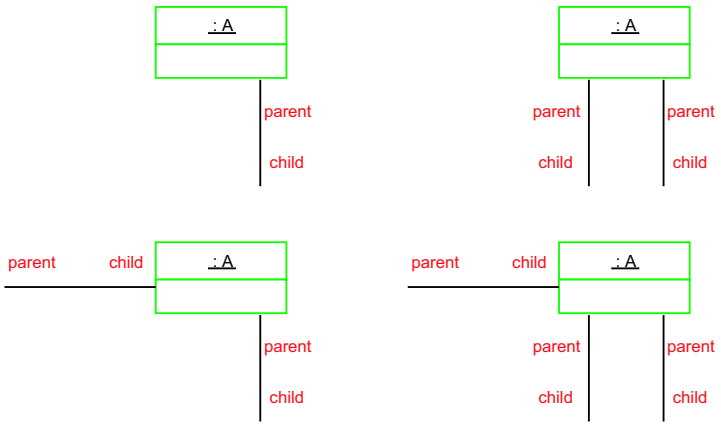


Abbildung 3.13: Die möglichen Konfigurationen der Klasse *A* des Metamodells *mm*

Es existieren also vier unterschiedliche Konfigurationen der Klasse *A*, die sich jeweils durch die Anzahl der unterschiedlichen Assoziationsenden mit denen sie verbunden sind unterscheiden. Abbildung 3.13 stellt die vier Konfigurationen graphisch dar. ❍

Ein Blick auf das Beispiel in Abbildung 3.13 macht deutlich, dass sich jede Instanzierung des Metamodells *mm* aus den dort abgebildeten Fragmenten zusammensetzen muss. Hierbei kann jedes dieser Fragmente beliebig oft innerhalb des Modells vorkommen. Besteht ein Modell ausschließlich aus solchen Fragmenten, so ist bereits sichergestellt, dass keine Multiplizitätsober- und -untergrenzen verletzt werden, da diese für jede einzelne Konfiguration eingehalten werden.

Allerdings müssen in einem gültigen Modell sämtliche Objektassoziationen genau zwei Objekte verbinden, d.h. „freischwebende“ Assoziationsenden sind nicht erlaubt. Damit dies sichergestellt werden kann, muss also für jede Klassenassoziation die Zahl seiner im Modell vorkommenden Objektassoziationsenden gleich groß sein. Diese Forderung wird nun durch das nachfolgende Instanzierbarkeits-Gleichungssystem formal festgehalten.

Definition 3.2.3 (Instanzierbarkeits-Gleichungssystem $IES(mm)$)
Es sei $mm \in \mathbb{MM}$ ein Metamodell, welches aus aus der Menge von Klassen $mm|_{CB} = \{class_0, \ldots, class_{|mm|_{CB}|-1}\}$ und der Menge von Assoziationen $mm|_{CA} = \{ASSO_0, \ldots, ASSO_{|mm|_{CA}|-1}\}$ besteht.
Für jede Klasse $class_l \in mm|_{CB}$ wird die Menge aller ihrer möglichen Konfigurationen angegeben als:

$$\{CF^0_{class_l}, \ldots, CF^{p_{class_l}}_{class_l}\} := configs(mm, class_l)$$

Dann existiert für jede Klassenassoziation $ASSO_i \in mm|_{CA}$ mit

$$\begin{aligned} ASSO_i &= \{end_j, end_k\} \text{ und} \\ CLASS_j &= \{class : end_j|_c \in types(class)\} \\ CLASS_k &= \{class : end_k|_c \in types(class)\} \end{aligned}$$

genau eine Gleichung der Form[3]:

$$\begin{aligned} EQ_{ASSO_i} : \quad & \sum_{\substack{class_j \in CLASS_j, \\ CF_l \in configs(mm, class_j)}} getCard(CF_l, ASSO_i, end_j) \cdot x_{class_j}^{CF_l} \\ = & \sum_{\substack{class_k \in CLASS_k, \\ CF_m \in configs(mm, class_k)}} getCard(CF_m, ASSO_i, end_k) \cdot x_{class_k}^{CF_m} \end{aligned}$$

Das *Instanzierbarkeits-Geleichungssystem* ($IES(mm)$) für ein Metamodell *mm* ist definiert als das folgende Gleichungssystem:

$$IES(mm) : \bigwedge_{ASSO \in mm|_{CA}} EQ_{ASSO}$$ ❍

Eine konkrete Lösung dieses Gleichungssystems gibt für jede Klasse an, wie oft jede Konfiguration dieser Klasse in einem Modell instanziert wurde. Dementsprechend bedeutet ein Wert *n* einer Unbekannten der Form $x_{class_l}^{CF_q}$, dass die Konfiguration cf_q der Klasse $class_l$ genau *n*-mal instanziert wurde. Alle Gleichungen des Gleichungssystems $IES(mm)$ enthalten keinen konstanten Anteil, d.h. eine Klasse kann immer entweder unendlich oft oder gar nicht instanziert werden.

Definition 3.2.4 (Instanzierbarkeit von Klassen (*instancable*$(CLASSES, mm)$))
Zu einem gegebenen Metamodell $mm \in \mathbb{MM}$ heißt eine Klasse $class \in mm|_{CB}$ *instanzierbar*, falls die Relation *instanceable* erfüllt ist, mit:

$$instancable(class, mm) :\Leftrightarrow \exists m \in \mathbb{M}_{mm} : class \in m|_{OB}|_{ot}$$ ❍

Die nachfolgende Definition legt fest, wann eine Konfiguration einer Klasse instanzierbar ist. Informell bedeutet dies, dass ein Modell als Instanz eines Metamodells Objekte der jeweiligen Klasse enthalten kann, welche genau die in der Konfiguration festgelegten Anzahlen von ein- bzw. ausgehenden Assoziationen haben.

Definition 3.2.5 (Instanzierbarkeit einer Konfiguration)
Sei $mm \in MM$ ein Metamodell und $class \in mm|_{CB}$ eine Klasse des Metamodells. Eine Konfiguration $CF_{class}^i \in configs(mm, class)$ der Klasse heißt *instanzierbar*, falls die Relation *instanceable* erfüllt ist, mit:

$$\begin{aligned} & instanceable(class, mm, CF_{class}^i) :\Leftrightarrow \\ & \quad \exists m \in \mathbb{M}_{mm}, o \in m|_{OB} : \\ & \qquad \forall cf \in CF_{class}^i : \exists oa \in m|_{OA} : \big(\quad o \in oa|_{OAE}|_o \wedge \\ & \qquad\qquad cf|_{ASSO} = oa|_{OAT} \wedge \\ & \qquad\qquad cf|_{end} \in oa|_{OAE} \wedge \\ & \qquad\qquad cf|_{card} = oa|_{card} \quad \big) \end{aligned}$$ ❍

[3] Falls gilt $|ASSO_i| = 1$, so impliziert dies $end_j = end_k$.

Mit Hilfe des folgenden Lemmas lässt sich die Instanzierbarkeit einer Konfiguration formal nachweisen:

Lemma 3.2.1 (Instanzierbarkeit einer Konfiguration)
Sei $mm \in \mathbb{MM}$ ein Metamodell, $class \in mm|_{CB}$ und $CF^i_{class} \in configs(mm, class)$ eine Konfiguration der Klasse *class*.

Die Konfiguration CF^i_{class} ist instanzierbar, falls für das Gleichungssystem $IES(mm)$ eine Lösung existiert für die

- alle Unbekannten einen endlichen, ganzzahligen Wert größer oder gleich 0 haben und
- die Unbekannte $x^{CF^i_{class}}_{class}$ einen Wert ungleich 0 hat.

(Beweis s. A.1.1, S. 279) ❑

Ist eine beliebige Konfiguration einer Klasse instanzierbar, so ist selbstverständlich auch die Klasse selbst instanzierbar. D.h. es existieren endliche, nicht leere Modelle, welche konform zu dem gegebenen Metamodell sind und Instanzen dieser Klasse enthalten. Lemma 3.2.2 stellt eine Technik für den Nachweis der Instanzierbarkeit von Klassen vor.

Lemma 3.2.2 (Instanzierbarkeit von Klassen)
Sei $mm \in \mathbb{MM}$ ein Metamodell und $class \in mm|_{CB}$. Die Klasse *class* ist instanzierbar, falls für das Gleichungssystem $IES(mm)$ eine Lösung existiert für die

- alle Unbekannten einen endlichen, ganzzahligen Wert größer oder gleich 0 haben und
- mindestens eine Unbekannte aus $\{x^{CF^i_{class}}_{class} : CF^i_{class} \in configs(mm, class)\}$ einen Wert ungleich 0 hat.

(Beweis s. A.1.1, S. 279) ❑

Aufbauend auf den beiden vorangegangen Lemmata wird nun eine Verifikationstechnik für die Instanzierbarkeit von Metamodellen vorgestellt.

Satz 3.2.1 (Instanzierbarkeit von Metamodellen)
Ein Metamodell $mm \in \mathbb{MM}$ ist instanzierbar, falls für das Gleichungssystem $IES(mm)$ eine Lösung existiert für die

- alle Unbekannten einen endlichen, ganzzahligen Wert größer oder gleich 0 haben und
- mindestens eine Unbekannte einen Wert ungleich 0 hat.

(Beweis s. A.1.1, S. 279) ❑

Für den Nachweis der Instanzierbarkeit eines Metamodells ist es nötig, die Lösbarkeit des diophantischen Gleichungssystems $IES(mm)$ nachzuweisen. Diophantische Gleichungssysteme sind Gleichungssysteme für die eine ganzzahlige Lösung gefordert ist. Darüber hinaus muss hier noch sichergestellt werden, dass für das Gleichungssystem eine Lösung existiert, bei der für sämtliche Variablen $0 \leq x^0_i, \ldots, x^{p_i}_i < \infty$ gilt.

Allgemein ist die Frage, ob ein diophantisches Gleichungssystem lösbar ist oder nicht, als das „Zehnte Hilbertsche Problem“ [Hil00] bekannt. Die „Fermatsche Vermutung“, welche besagt, dass dies für lineare Gleichungen möglich ist, wurde mittlerweile bewiesen [Wil95, TW95]. Für lineare diophantische Gleichungssystem der Art wie sie hier auftreten existieren also Lösungsverfahren.

Generell lassen sich lineare diophantische Gleichungssysteme in polynomialer Zeit lösen [Sch86]. In [AHL00] findet sich ein Algorithmus, der auch diophantische Gleichungssysteme mit Unter- und Obergrenzen für die Lösungen lösen kann. Mit Hilfe eines solchen Algorithmus ist es möglich, gezielt nach nichtnegativen Lösungen zu suchen.

Problematisch sind Assoziationen mit einem $*$, da sie zu unendlich vielen Konfigurationen einer Klasse und somit unendlichen vielen Unbekannten im Gleichungssystem führen. Eine denkbare Lösung dieses Problems ist es, für den Nachweis der Instanzierbarkeit eines Metamodells den $*$ in Klassenassoziationen durch einen berechenbaren Wert zu ersetzen. Ist das somit restriktivere Metamodell instanzierbar, so gilt dies augenscheinlich auch für ein Metamodell mit $*$-Assoziationen.

Das nachfolgende Beispiel veranschaulicht exemplarisch, wie sich mit Hilfe von Satz 3.2.1 die Instanzierbarkeit eines Metamodells nachweisen lässt.

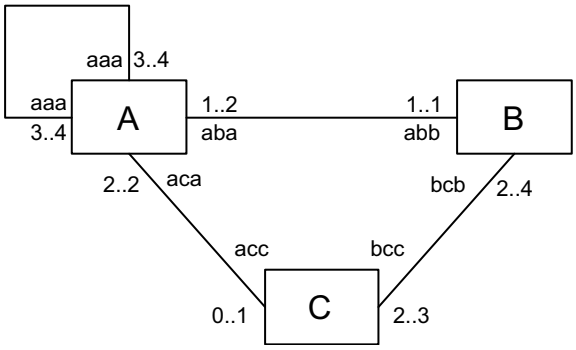


Abbildung 3.14: Beispiel für ein instanzierbares Metamodell

Beispiel: Abbildung 3.14 zeigt ein Metamodell dessen Instanzierbarkeit im weiteren überprüft werden soll. Das Metamodell besteht aus drei Klassen, die jeweils durch Assoziationen mit verschiedenen Multiplizitäten verbunden sind. Zusätzlich existiert eine symmetrische Assoziation an der Klasse A. Die Namen von Klassenassoziationen setzen sich im Beispiel jeweils aus den Namen der Klassen, welche sie verbinden, zusammen. So heißt beispielsweise die Assoziation zwischen den Klassen A und B ab. Assoziationsenden leiten sich aus dem Assoziationsnamen gefolgt vom Namen der Klasse mit der sie verbunden sind ab. Das Assoziationsende der Klassenassoziation ab an der Klasse A heißt dementsprechend aba. Zur besseren Verständlichkeit werden bei den in Abbildung 3.14 dargestellten Assoziationen die Namen der Assoziationsenden als Rollenname verwendet.

Im Folgenden sind die verschiedenen möglichen Konfigurationen der drei Klassen aufgezählt. So existieren für die Instanzen der Klasse C beispielsweise drei mögliche Konfigurationen, für die jeweils die Kardinalitäten von zwei Assoziationen berücksichtigt werden müssen: Die ersten Konfiguration steht für C-Objekte mit genau zwei ausgehenden Assoziation zu genau einem B-Objekt, die zweite für solche mit drei und die dritte für vier solche Assoziationen. Alle C-Objekte haben genau zwei ausgehende Assoziationen des Typs ac zu Objekten des Typs A, wie im Metamodell gefordert.

$$
\begin{aligned}
configs(mm,A) = \{\ & \{(ab,aba,1),(ac,aca,0),(aa,aaa,3)\}, && (\hat{=}x_a^0)\\
& \{(ab,aba,1),(ac,aca,1),(aa,aaa,3)\} && (\hat{=}x_a^1)\\
& \{(ab,aba,1),(ac,aca,0),(aa,aaa,4)\}, && (\hat{=}x_a^2)\\
& \{(ab,aba,1),(ac,aca,1),(aa,aaa,4)\} && (\hat{=}x_a^3)\\
\}&
\end{aligned}
$$

$$
\begin{aligned}
configs(mm,B) = \{\ & \{(ab,abb,1),(bc,bcb,2)\}, && (\hat{=} x_b^0)\\
& \{(ab,abb,2),(bc,bcb,2)\}, && (\hat{=} x_b^1)\\
& \{(ab,abb,1),(bc,bcb,3)\}, && (\hat{=} x_b^2)\\
& \{(ab,abb,2),(bc,bcb,3)\} && (\hat{=} x_b^3)\\
\}\\
configs(mm,C) = \{\ & \{(ac,acc,2),(bc,bcc,2)\}, && (\hat{=} x_c^0)\\
& \{(ac,acc,2),(bc,bcc,3)\}, && (\hat{=} x_c^1)\\
& \{(ac,acc,2),(bc,bcc,4)\} && (\hat{=} x_c^2)\\
\}
\end{aligned}
$$

Gemäß Satz 3.2.1 wird nun das Gleichungssystem IES gebildet. Die linke Seite der ersten Gleichung EQ_{ab} berechnet die Anzahl aller Assoziationsenden vom Typ aba. Dabei bezeichnen die Unbekannten x_a^0 und x_a^1 die Anzahl der A-Objekte in einem beliebigen Modell die zu der Konfiguration von A mit bzw. ohne die Assoziation zu einem C-Objekt gehören. Da beide Konfigurationen jeweils über eine ausgehenden Assoziation des Typs ab verfügen wird jeweils mit 1 multipliziert.

Die rechte Seite von EQ_{ab} berechnet die mögliche Anzahl von abb-Assoziationsenden in Abhängigkeit davon wieviele Instanzen der vier möglichen B-Konfigurationen auftreten. Die Aussage der Gleichung ist somit, dass die Anzahl der abb- und aba-Enden gleich sein muss, da jede Assoziation des Typs ab genau zwei solcher Enden hat. EQ_{bc} und EQ_{ac} machen analoge Aussagen für die Klassenassoziationen bc und ac.

$$
\begin{aligned}
EQ_{ab}: &\quad 1\cdot x_a^0+1\cdot x_a^1+1\cdot x_a^2+1\cdot x_a^3 = 1\cdot x_b^0+2\cdot x_b^1+1\cdot x_b^2+2\cdot x_b^3\\
EQ_{bc}: &\quad 2\cdot x_b^0+2\cdot x_b^1+3\cdot x_b^2+3\cdot x_b^3 = 2\cdot x_c^0+3\cdot x_c^1+4\cdot x_c^2\\
EQ_{ac}: &\quad 0\cdot x_a^0+1\cdot x_a^1+0\cdot x_a^2+1\cdot x_a^3 = 2\cdot x_c^0+2\cdot x_c^1+2\cdot x_c^2\\
EQ_{aa}: &\quad 3\cdot x_a^0+3\cdot x_a^1+4\cdot x_a^2+4\cdot x_a^3 = 3\cdot x_a^0+3\cdot x_a^1+4\cdot x_a^2+4\cdot x_a^3
\end{aligned}
$$

Die symmetrische Klassenassoziation aa erzeugt eine triviale Gleichung, die immer wahr ist. Dies entspricht der Intuition, da symmetrische Assoziationen nie der Instanzierbarkeit eines Metamodells im Wege stehen können. Insgesamt ergibt sich der folgende (unendliche) Lösungsraum für das Gleichungssystem:

$$
\begin{aligned}
& x_a^0 = 2x_a^2+x_b^1-\frac{1}{2}x_b^2+\frac{1}{2}x_b^3+x_c^0-\frac{1}{2}x_c^1 \in \mathbb{N}_0^+ && x_b^2 \in \mathbb{N}_0^+\\
& x_a^1 = -x_a^3+2x_c^0+2x_c^1+2x_c^2 \in \mathbb{N}_0^+ && x_b^3 \in \mathbb{N}_0^+\\
& x_a^2 \in \mathbb{N}_0^+ && x_c^0 \in \mathbb{N}_0^+\\
& x_a^3 \in \mathbb{N}_0^+ && x_c^1 \in \mathbb{N}_0^+\\
& x_b^0 = -x_b^1-\frac{3}{2}x_b^2-\frac{3}{2}x_b^3+x_c^0+\frac{3}{2}x_c^1+2x_c^2 \in \mathbb{N}_0^+ && x_c^2 \in \mathbb{N}_0^+\\
& x_b^1 \in \mathbb{N}_0^+
\end{aligned}
$$

Eine mögliche Lösung mit ganzzahligen, nichtnegativen Werten lautet demnach:

$$x_a^0 = 0 \qquad x_b^0 = 2 \qquad x_c^0 = 1$$
$$x_a^1 = 5 \qquad x_b^1 = 1 \qquad x_c^1 = 1$$
$$x_a^2 = 0 \qquad x_b^2 = 0 \qquad x_c^2 = 1$$
$$x_a^3 = 1 \qquad x_b^3 = 1$$

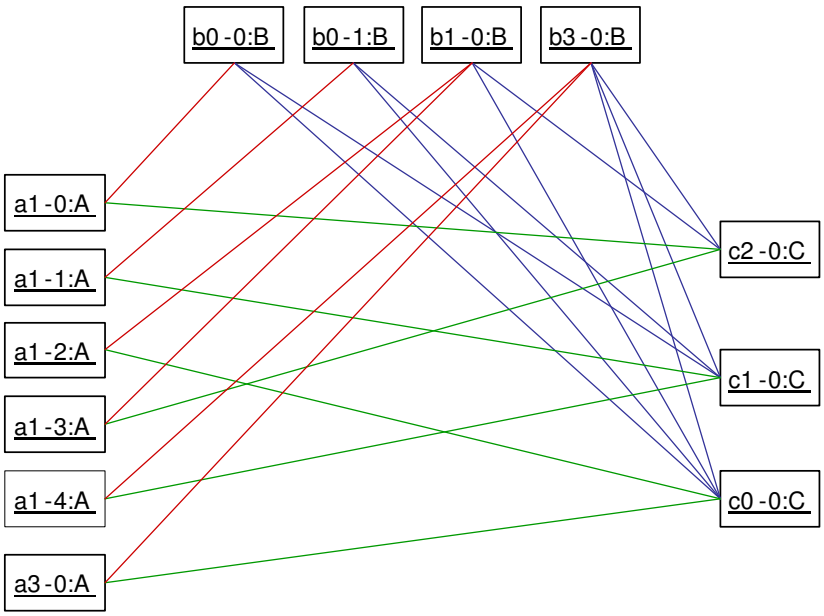


Abbildung 3.15: Eine gültige Instanz des in Abbildung 3.14 dargestellten Metamodells

Abbildung 3.15 stellt ein Objektgeflecht dar, in dem Objekte Konfigurationen entsprechend dieser Lösung aufweisen. Objekte der Konfiguration x_a^1 sind beispielsweise durch einen Objektidentifikator der mit $a1$ beginnt gekennzeichnet. Die Assoziationen des Typs *aaa* zwischen Objekten des Typs *A* wurden zur besseren Übersichtlichkeit nicht dargestellt. ❍

3.3 BOTL Regelwerke

BOTL spezifiziert Modelltransformationen anhand von Transformationsregeln. Jede Regel definiert die Abbildung von Fragmenten des Ursprungsmodells auf neu zu erzeugende Fragmente des Zielmodells. Eine Regel besteht aus einer Ursprungs- und einer Zielmodellvariable, welche die Struktur und den Inhalt der gesuchten und erzeugten Modellfragmente festlegen. Im Weiteren wird der Aufbau von BOTL-Regeln und BOTL-Regelwerken formal definiert.

Innerhalb von Modellvariablen werden Terme zur Beschreibung der Beziehungen zwischen den Identifikatoren und Attributen der Ursprungs- und Zielmodellfragmente verwendet. BOTL enthält keine umfassende Definition für korrekte Terme und ihrer Semantik. Hierzu existiert bereits eine Vielzahl von Arbeiten, auf die an dieser Stelle verwiesen wird.

Zunächst wird die Menge aller Variablen eingeführt. Variablen sind eindeutig von Identifikatoren und konstanten Werten unterscheidbar.

Definition 3.3.1 (Variablen $\mathbb{VAR}$)
$\mathbb{VAR}$ bezeichnet die Menge aller möglichen Variablennamen. Es gilt:

$$|\mathbb{VAR}| = \infty \wedge \mathbb{VAR} \cap \mathbb{ID} = \emptyset \wedge \mathbb{VAR} \cap \mathbb{T}|_{T_{\text{val}}} = \emptyset \qquad \text{❍}$$

Terme werden verwendet, um den Wert von erzeugten Objektattributen zu berechnen. Definition 3.3.2 definiert die Menge aller Terme über Typen.

Definition 3.3.2 (Terme $Term_{TA}$)
$Term_{TA}$ bezeichnet die Menge aller korrekten Terme über Typen der Typbelegung *TA*. $Term_{TA}$ enthält für jedes beliebige *TA* immer auch den Wert $\diamondsuit$. Die Menge der möglichen Typen eines Terms wird bestimmt durch die Abbildung:

$$type : Term_{TA} \rightarrow \mathscr{P}(TA) \qquad \text{❍}$$

Bemerkung: Der Wert $\diamondsuit$ dient innerhalb von BOTL als ein Art Jocker-Symbol. Seine exakte Semantik ist in Abschnitt 3.5 festgelegt. In der graphischen Darstellung von Regeln wird statt $\diamondsuit$ das Symbol „♯“ verwendet (siehe Abschnitt 3.4), da diese Form der Darstellung einfacher durch eine Werkzeugunterstützung abbildbar ist.

Identifikatorterme werden verwendet, um die Identität erzeugter Objekte zu berechnen. Die Menge aller Identifikatorterme wird im Folgenden definiert:

Definition 3.3.3 (Identifikatorterme $Term_{\mathbb{ID}}$)
$Term_{\mathbb{ID}}$ bezeichnet die Menge aller gültigen Terme über der Menge der Identifikatoren $\mathbb{ID}$. Dementsprechend können die Elemente von $Term_{\mathbb{ID}}$ aus

- *Konstanten* (d.h. ein Elemente aus $\mathbb{ID}$),
- *Variablen* (d.h. ein Elemente aus $\mathbb{VAR}$),
- *n-Tupeln* aus Variablen oder Konstanten mit $n \in \mathbb{N}^+$ und
- dem Wert $\diamondsuit$

bestehen. Für den Fall, dass ein Identifikator aus einem Tupeln von Variablen und Konstanten besteht, muss eine bijektive Abbildung *uK* existieren, die die Tupelwerte auf $\mathbb{ID}$ abbildet. *uK* generiert Identifikatoren die sich von allen Konstanten, Variablen und $\diamondsuit$ unterscheiden. ❍

Beispiele für gültige Identifikatorterme sind: $\diamondsuit$, *id*, (id), (cid, aid).

Objektvariablen beschreiben zu findende Ursprungs-Objekte oder zu erzeugende Ziel-Objekte. Ihre graphische Darstellung ähnelt der von Objekten in UML-Objektmodellen (siehe Abbildung 3.16). Sie enthalten jedoch Terme anstelle der Objektidentifikatoren und Attributwerte. Zur besseren Lesbarkeit werden Primärschlüsselattribute im Objektidentifikator aufgelistet (siehe Abschnitt 3.4) oder wie in Abbildung 3.16 in farblich abgesetzter Schrift dargestellt.

Definition 3.3.4 (Objektvariable *ov*)
Eine *Objektvariable ov* ist ein Tupel $(ovid, oiv, otv, VV)$ aus

- einem eindeutigen *Objektvariablenidentifikator ovid* $\in \mathbb{ID}$,

- einem *Objektvariablenidentifikatorterm* $oiv = \begin{cases} term \in Term_{\mathbb{ID}} & \text{falls } ov|_{otv}|_{Keys} = \emptyset \\ \varepsilon & \text{sonst} \end{cases}$
- einem *Objekttyp* $otv \in \mathbb{C}$ und
- einer Menge *Objektvariablenattributen* VV aus Tupeln (a,t) mit:

$$\begin{aligned} \forall (a,t) \in VV: \quad & a \in \mathbb{ID}, t \in Term_{TA} \\ & \wedge\, otv|_A|_n = VV|_a \\ & \wedge\, \left|otv|_A\right| = \left|VV\right| \\ & \wedge\, \forall (a,t) \in VV: \quad \exists (a,p) \in otv|_A \wedge p \in type(t) \end{aligned} \tag{3.17}$$

Die Menge aller denkbaren Objektvariablen wird mit $\mathbb{OV}$ bezeichnet. Die Relation *consistent* gilt, falls eine Objektvariable konsistent zu einem gegebenen Metamodell ist, d.h.

$$consistent(ov, mm) :\Leftrightarrow ov|_{otv} \in mm|_{CB}$$

Um leichter auf Attributterme zugreifen zu können, wird (analog zu Def. 3.1.11) die folgende Notation eingeführt:

$$ov.att := \begin{cases} t & \text{falls } \exists a : (a,t) \in ov|_{VV} \wedge a = att \\ \perp & \text{sonst} \end{cases}$$ ❍

Der Objektvariablenidentifikator *ovid* ist üblicherweise nicht sichtbar. Er wird jedoch benötigt, um verschiedenen Objektvariablen mit dem gleichen Inhalt (abgesehen von ihren *ovid*-Werten) innerhalb einer Menge von Objektvariablen unterscheiden zu können (siehe auch Def. 3.3.5). Der Identifikatorterm einer Objektvariablen kann auch den Wert $\diamondsuit$ haben. In einem Quellmodell matcht eine solche Objektvariable auf Objekte mit beliebigem Identifikator, während aus einer solchen Objektvariable im Zielmodell ein Objekt mit einem generierten, eindeutigen Identifikator erzeugt wird. Eine formale Definition der Semantik des $\diamondsuit$-Wertes als Objektvariablenidentifikatorterm wird in Abschnitt 3.5 angegeben.

(3.17) legt fest, dass für jedes Attribut einer Objektvariablen ein Term spezifiziert werden muss, dessen Typ zum Typ des Attributs passt.

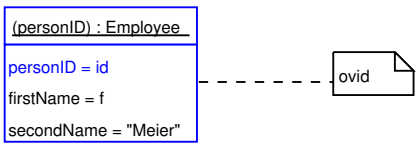


Abbildung 3.16: Eine Objektvariable

Beispiel: Abbildung 3.16 zeigt ein Beispiel für eine Objektvariable, deren Typ aus dem Metamodell mm_α aus Abbildung 3.5 (S. 57) stammt. Konstante Werte sind entweder Zahlenwerte oder Zeichenketten in Anführungszeichen, für Attribute vom Typ String wie hier für das Attribut secondName. Für den Identifikator wird im Beispiel kein Term angegeben, da die Klasse Employee über ein Primärschlüsselattribut personID verfügt, welches in der Objektvariable farblich abgesetzt dargestellt ist.

Zusätzlich werden alle Primärschlüsselattribute in Klammern im Objektvariablenidentifikator aufgelistet. Um eine Objektvariable referenzieren zu können, wird ihr Namen in Form eines Kommentars an sie angefügt.

Innerhalb des BOTL-Formalismus wird die dargestellte Objektvariable durch das Tupel

$$(ovid, \varepsilon, Emp, \{(personID, id), (firstName, f), (secondName, \text{„}Meier\text{“})\})$$

repräsentiert. Die Klasse Emp wird selbst wieder durch ein Tupel repräsentiert, welches in dem auf Seite 53 vorgestellten Beispiel angeführt ist. ❍

Ähnlich wie innerhalb einer Objektbelegung sichergestellt ist, dass keine Objekte mit gleichem Typ und gleicher Identität existieren, wird innerhalb einer Objektvariablenbelegung gefordert, dass sämtliche Objektvariablen über paarweise unterschiedliche Objektvariablenidentifikatoren verfügen.

Definition 3.3.5 (Objektvariablenbelegung *OVB*)
Eine *Objektvariablenbelegung* $OVB \in \mathscr{P}(\mathbb{OV})$ ist eine endliche Menge von Objektvariablen, für die gilt:

$$\forall ov_i, ov_j \in OVB : (ov_i|_{ovid} = ov_j|_{ovid} \Rightarrow ov_i = ov_j)$$

Eine Objektvariablenbelegung OVB ist genau dann konsistent zu einem gegebenen Metamodell $mm \in \mathbb{MM}$, wenn die im Folgenden definierte Relation *consistent* gilt. ist:

$$consistent(OVB, mm) :\Leftrightarrow \forall ov \in OVB : consistent(ov, mm)$$ ❍

Analog zu Objektvariablen dienen Objektvariablenassoziationen als Platzhalter für Objektassoziationen. Graphisch werden sie genauso wie Objektassoziationen dargestellt.

Definition 3.3.6 (Objektvariablenassoziation *ova*)
Eine *Objektvariablenassoziation ova* ist eine Tupel $(OVAT, cardv, OVAE)$, bestehend aus

- einer Klassenassoziation $OVAT \in \mathbb{AE}$,
- einer *Kardinalität* $cardv \in \mathbb{N}$ und
- einer Menge $OVAE$ von *Objektvariablenassoziationsenden* $ovae = (ae, ov)$, bestehend aus
  - einem Klassenassoziationsende $ae \in OVAT$ und
  - einer Objektvariable $ov \in \mathbb{OV}$

  für die gilt:

$$\begin{aligned} &\{ae : (ae, ov) \in OVAE\} = OVAT \\ \wedge\ &\forall ovae \in OVAE : ovae|_{ae}|_c \in types(ovae|_{ov}|_{otv}) \\ \wedge\ &1 \leq |OVAE| \leq 2 \end{aligned}$$

Eine Objektvariablenassoziation ist genau dann konsistent zu einer gegebenen Objektvariablenbelegung, wenn die im Folgenden definierte Relation *consistent* gilt:

$$consistent(ova, OVB) :\Leftrightarrow ova|_{ov} \in OVB$$

$\mathbb{OVA}$ bezeichnet die Menge aller denkbaren Objektvariablenassoziationen. ❍

Die Abbildung $getova_{mv}$ wird eingeführt, um den Umgang mit Objektvariablenassoziationen zu erleichtern. Sie liefert zu zwei Enden jeweils die passende Objektvariablenassoziation innerhalb einer Modellvariable, falls eine solche Assoziation existiert.

Definition 3.3.7 ($getova_{mv}(ovae_0, ovae_1)$)
Zu einer gegebenen Modellvariable $mv = (mm, OVB, OVA)$ sei

$$getova_{mv} : OVA|_{OVAE} \times OVA|_{OVAE} \to OVA \cup \{\bot\}$$

$$getova_{mv}(ovae_0, ovae_1) \mapsto \begin{cases} ova & \text{falls } \exists ova \text{ mit } \{ovae_0, ovae_1\} = ova|_{OVAE} \\ \bot & \text{sonst} \end{cases}$$

❍

getova liefert genau die Objektvariablenassoziation, welche die gegebenen Objektvariablenassoziationsenden enthält. Existiert keine Objektvariablenassoziation mit den angegeben Ende (bzw. dem angegebenen Ende), so liefert *getova* den Fehlerwert $\bot$.

Modellvariablen bilden jeweils die linke und rechte Seite von BOTL-Regeln. Eine Modellvariable besteht aus Objektvariablen, die über Objektvariablenassoziationen verbunden sein können. Somit beschreiben Modellvariablen die Strukturen nach denen in Quellmodellen gesucht wird, bzw. die Strukturen die durch BOTL-Regeln erzeugt werden.

Definition 3.3.8 (Modellvariable *mv*)
Eine *Modellvariable mv* ist ein Tupel (mm, OVB, OVA) bestehend aus

- einem Metamodell $mm \in \mathbb{MM}$,
- einer Objektvariablenbelegung *OVB* mit $consistent(OVB, mm)$ und
- einer endlichen Menge von Objektvariablenassoziationen *OVA* mit $\forall ova \in OVA : consistent(ova, OVA)$.

$\mathbb{MV}$ bezeichnet die Menge aller denkbaren Modellvariablen. ❍

Eine Modellvariable kann also Konsistenzbedingungen, die durch ihr Metamodell definiert werden verletzen, da die Kardinalitäten der Objektvariablenassoziation nicht mit den im Metamodell definierten Multiplizitäten der Klassenassoziationen konform sein müssen.

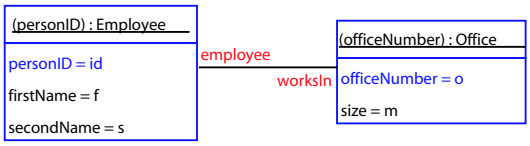


Abbildung 3.17: Eine Modellvariable

Beispiel: Abbildung 3.17 stellt ein Beispiel für eine Modellvariable aus zwei Objektvariablen, die über eine Objektvariablenassoziation verbunden sind, dar. Das Metamodell der Modellvariablen ist das in Abschnitt 3.1 vorgestellte Metamodell mm_α. ❍

Modelltransformationsregeln spezifizieren, wie aus Teilen von Quellmodellen neue Teile eines Zielmodells erzeugt werden. Die Art der Modellteile nach denen in einem Quellmodell gesucht wird und die Art der neu erzeugten Modellteile werden durch jeweils eine Modellvariable spezifiziert. Dementsprechend besteht eine Modelltransformationsregel aus genau zwei Modellvariablen:

Definition 3.3.9 (Modelltransformationsregel r)
Eine *Modelltransformationsregel* (oder kurz: *Regel*) r ist ein Tupel (mv_0, mv_1), bestehend aus

- einer *Quellmodellvariable* $mv_0 \in \mathbb{MV}$ und
- einer *Zielmodellvariable* $mv_1 \in \mathbb{MV}$.

$\mathbb{R}$ bezeichnet die Menge aller denkbaren Modelltransformationsregeln. Die Menge aller denkbaren Regeln mit einem Quellmetamodell $mm_0 \in \mathbb{MM}$ und einem Zielmetamodell $mm_1 \in \mathbb{MM}$ wird mit $\mathbb{R}_{mm_0,mm_1}$ bezeichnet. ❍

Die beiden Modellvariablen werden graphisch durch einen Pfeil getrennt, der von der Quellmodellvariable auf die Zielmodellvariable verweist. Zur besseren Verständlichkeit wird die Quellmodellvariable im Weiteren auch als *linke Seite* und die Zielmodellvariable als *rechte Seite der Regel* bezeichnet.

Ein $\diamondsuit$ als Attributwert auf der rechten Regelseite führt zu einem Modellfragment das ebenfalls einen $\diamondsuit$ als Attributwert im jeweiligen Attribut hat. Dies bedeutet, dass der Wert des Attributs noch undefiniert ist. Werden mehrere Modellfragmente verschmolzen, so wird der Wert $\diamondsuit$ durch jeden beliebigen anderen Wert überschrieben.

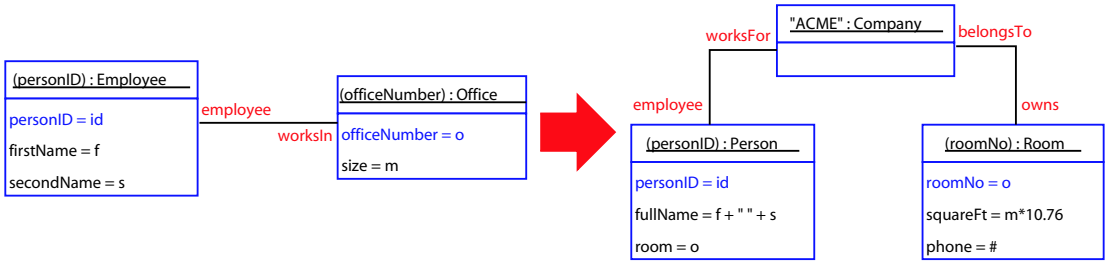


Abbildung 3.18: Die Modelltransformationsregel r_0

Beispiel: Abbildung 3.18 stellt ein Beispiel für eine Regel zur Transformation von Instanzen des Metamodells aus Abbildung 3.5 in Instanzen des Metamodells aus Abbildung 3.6 dar. Die Regel ist ein Element der Menge $\mathbb{R}_{mm_\alpha,mm_\beta}$, d.h. sie bildet Instanzen des Metamodells mm_α auf solche des Metamodells mm_β ab. In diesem Fall erzeugt die Regel für jedes Employee-Objekt, das mit einem Objekt des Typs Office verbunden ist, jeweils ein Objekt des Typs Person bzw. Room, die beide mit einem Company-Objekt assoziiert sind. ❍

Ein Modelltransformationsregelwerk wird nun als eine Menge von Regeln definiert. Bei der Anwendung eines Regelwerks werden sämtliche enthaltene Regeln angewendet (siehe auch Abschnitt 3.5). Da BOTL verschiedene Quellmodelle auf ein Zielmodell abbilden kann, können die einzelnen Regeln unterschiedliche Quellmetamodelle haben. Zu jedem Quellmetamodell existiert jedoch genau ein Quellmodell. Damit alle Regeln dasselbe Zielmodell erzeugen, müssen sämtliche Regeln eines Regelwerks dasselbe Ziel-(meta)-Modell vorweisen.

Definition 3.3.10 (Modelltransformationsregelwerk R)
Ein *Modelltransformationsregelwerk* (oder kurz: Regelwerk) $R \in \mathscr{P}(\mathbb{R})$ ist eine Menge von Modelltransformationsregeln, für die gilt:

$$\forall r_i, r_j \in R \text{ mit } r_i \neq r_j : r_i|_{mv_1}|_{mm} = r_j|_{mv_1}|_{mm} \wedge \tag{3.18}$$

$$\forall r \in R : r|_{mv_0}|_{mm} \neq r|_{mv_1}|_{mm} \tag{3.19}$$

Die Menge aller denkbaren Modelltransformationsregelwerke wird mit $\mathbb{RW}$ bezeichnet. ❍

Aussage (3.18) legt fest, dass alle Regeln eines Modelltransformationsregelwerks Abbildungen mit demselben Zielmetamodell definieren. (3.19) besagt, dass keine Regel das Zielmetamodell als Quellmetamodell haben darf. Da in BOTL jeweils nur ein Modell zu jedem Metamodell existieren darf (auch wenn mehrere Metamodelle bis auf ihren Identifikator identisch sein dürfen), werden so Transformationen des Ursprungsmodells (sog. Rewrite-Transformationen) ausgeschlossen. Mehrere Quellmodelle mit gleichem Metamodell lassen sich durch die Verwendung isomorpher Metamodelle gemäß Definition 3.1.10 (siehe S. 57) realisieren. In Abschnitt 3.6 wird eine entsprechende Erweiterung des BOTL-Formalismus für Rewrite-Transformationen vorgestellt.

Um leicht auf die Quell- und Ziel(meta)modelle eines Regelwerks zugreifen zu können, werden in der folgenden Definition vereinfachte Schreibweisen für sie eingeführt:

Definition 3.3.11 (Quell-/Zielmetamodelle, Menge aller möglichen Quell-/Zielmodelle)
Es sei $R \in \mathbb{RW}$ ein Regelwerk. Die Menge

$$MM_R^0 := \bigcup_{r \in R} r|_{mv_0}|_{mm}$$

bezeichnet die Menge aller *Quellmetamodelle* des Regelwerks *R*.

$$mm_R^1 := mm \in \mathbb{MM} \text{ mit } R|_{mv_1}|_{mm} = \{mm\}$$

ist das *Zielmetamodell* des Regelwerks *R*.

Weiter ist

$$\mathbb{M}_R^0 := \{M \in \mathscr{P}(\mathbb{M}) : mm \in MM_R^0 \Leftrightarrow |\mathbb{M}_{mm} \cap M| \leq 1\}$$

die Menge aller möglichen *Mengen von Quellmodellen* für das Regelwerk *R* und

$$\mathbb{M}_R^1 := \mathbb{M}_{mm_R^1}$$

die Menge aller möglichen *Zielmodelle* von *R*. ❍

Jedes Element der Menge $\mathbb{M}_R^0$ ist also eine Menge von Modellen. Wie sich im Verlauf der Arbeit zeigen wird, werden im Verlauf einer Modelltransformation nicht vorhandene Modelle und Modelle deren Metamodell kein Quellmetamodell ist ignoriert. Dementsprechend enthält jedes Element von $\mathbb{M}_R^0$ für jedes Quellmetamodell *mm* des Regelwerks *höchstens* ein Modell dieses Typs *mm*.

Beispiel: Abbildung 3.19 zeigt eine weitere Regel $r_1 \in \mathbb{R}_{mm_alpha,mm_\beta}$ mit dem gleichen Quell- und Zielmetamodell wie die Regel r_0 von Seite 75. Die Regel sucht nach Office-Objekten, die mit einem Phone-Objekt assoziiert sind und trägt den jeweils gefundenen Wert des Attributs phoneNumber in das Attribut phone des entsprechenden Room-Objekts ein. Hierbei wird die interne Telefonnummern noch um eine nationale und internationale Vorwahl ergänzt. Implizit wird der Wert vom Typ int in den Typ String konvertiert.

Zusammen mit der Regel r_0 bilden die beiden Regeln das Regelwerk $R := \{r_0, r_1\}$. Offensichtlich gilt:

$$MM_R^0 = \{mm_\alpha\}$$
$$mm_R^1 = mm_\beta$$
$$m_\alpha \in \mathbb{M}_R^0$$

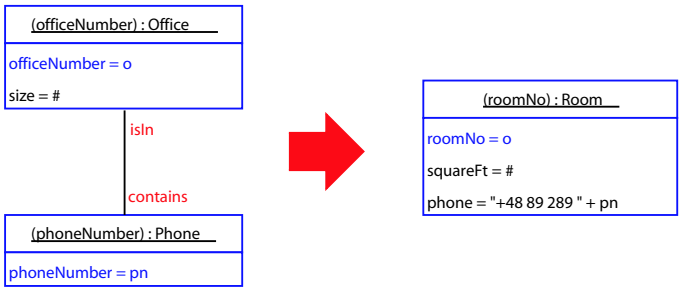


Abbildung 3.19: Die Modelltransformationsregel r_1

❍

Um innerhalb einer Transformationsspezifikation Informationen zur Verfügung zu stellen, die nicht im Quellmodell enthalten sind, oder um die Auswahl von gefunden Quellfragmenten für eine Transformation in geeigneter Weise einzuschränken, ist oftmals die Angabe konstanter Werte in Transformationsregeln notwendig.

Konstante Werte in Regeln sind jedoch für die Wiederverwendung von Transformationsregeln in unterschiedlichen Kontexten vielfach hinderlich, da sie eine Anpassung der Regeln für den gegebenen Anwendungsfall notwendig machen. Der konstante Wert „ACME“ in der in Abbildung 3.18 (Seite 75) dargestellten Regel ist ein Beispiel hierfür. So werden für die Erzeugung neuer Modellelemente häufig zusätzliche Informationen, wie hier ein Name, benötigt. Um den Umgang mit solchen Regeln zu vereinfachen und die Wiederverwendbarkeit von Regelwerken zu verbessern, werden im Folgenden *parametrisierbare* Regeln als Kurzschreibweise für eine potentiell unendliche Menge von Regeln eingeführt.

Definition 3.3.12 (Parametrisierbare Regeln)
Eine parametrisierbare Regel $r(\$var_0, \ldots, \$var_n)$ enthält in Identifikatoren und/oder Attributvariablen von Objektvariablen Paramtervariablen. Sie repräsentiert sämtliche mögliche Regeln, die sich durch die Belegung der Parametervariablen durch konstante Werte ergeben.

Durch Angabe konstanter Werte für die Parameter können Teilmengen oder einzelne Regeln aus der Menge der durch eine parametrisierbare Regel repräsentierten Regeln referenziert werden. ❍

Abbildung 3.20: Die paramterisierbare Regel $r(\$NAME, \$BOOL)$

Abbildung 3.20 zeigt die paramtrisierbare Regel $r(\$NAME, \$BOOL)$. Diese repräsentiert eine potentiell unendlich große Menge von Regeln.

Beispiel: $r(\text{„ACME“}, \$BOOL)$ ist eine verkürzte Schreibweise für die beiden in Abbildung 3.21 dargestellten Regeln. Während der Wert $NAME fest mit dem Wert „ACME“ vorbelegt ist, existiert für jeden möglichen Wert von $BOOL eine eigene Regel. Das Attribut *att* ist vom Typ *boolean* für den nur zwei mögliche Werte (*true* und $false$) definiert sind. Dementsprechend repräsentiert diese Regel genau zwei konkrete Regeln. ❍

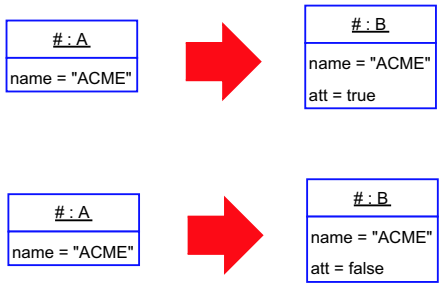


Abbildung 3.21: Die durch r(„ACME“, $BOOL) repräsentierten BOTL-Regeln

Um ein BOTL-Regelwerk „umgekehrt“ anwenden zu können, wird an dieser Stelle der Begriff *Umkehrregel* definiert. Die Umkehrregel einer BOTL-Regel erhält man durch Vertauschen von Quell- und Zielmodellvariable:

Definition 3.3.13 (Umkehrregel (r^{-1}))
Sei $r = (mv_0, mv_1) \in \mathbb{R}$ eine Regel. Dann ist r^{-1} die *Umkehrregel* von r, wobei gilt:

$$r^{-1} := (mv_1, mv_0) \qquad \text{❍}$$

Dementsprechend setzt sich ein Umkehrregelwerk aus den Umkehrregeln eines Regelwerks zusammen. Da ein BOTL-Regelwerk jedoch generell eine Menge von Modellen in ein Zielmodell überführt kann für die Bijektivität eines Regelwerks jeweils nur eines der Quellmetamodelle berücksichtigt werden, falls mehr als ein solches Quellmetamodell existieren sollte. Dementsprechend ist ein Regelwerk immer bijektiv bezüglich eines Quellmetamodells. Das Umkehrregelwerk bezüglich dieses Quellmetamodells enthält also die Umkehrregeln aller Regeln des Regelwerks mit diesem Quellmetamodell.

Definition 3.3.14 (Umkehrregelwerk (R^{-1}_{mm}))
Sei $R \in \mathbb{RW}$ ein Regelwerk und $mm \in \mathbb{MM}^0_R$ ein Quellmetamodell. Dann ist das *Umkehrregelwerk* R^{-1}_{mm} bezüglich des Quellmetamodells mm das Regelwerk für das gilt:

$$R^{-1}_{mm} := \{r^{-1} : r \in R \wedge r|_{mv_0}|_{mm} = mm\} \qquad \text{❍}$$

Eine interessante, mögliche Eigenschaft von Regelwerken ist ihre Bijektivität bezüglich eines Quellmodells. Diese Eigenschaft ist erfüllt, falls die Anwendung des Umkehrregelwerks bezüglich des Quellmetamodells für ein, durch das Regelwerk erzeugtes, Modell wieder das ursprüngliche Quellmodell liefert. Innerhalb von Abschnitt 4.3 (S. 167 ff) wird diese Eigenschaft eingehender diskutiert.

3.4 Ein UML-Profil für BOTL-Regelwerke

In Abschnitt 3.3 wurde bereits exemplarisch eine UML-basierte graphische Notation für BOTL-Metamodelle und -Regeln verwendet. Im Folgenden wird ein UML-Profil definiert mit dem sich BOTL-Metamodelle und -Regeln auf Basis von Klassen- bzw. Objektdiagrammen darstellen lassen. Hierzu wird die UML um zwei neue Typen von Tagged Values (siehe Tabelle 3.2) und einen neuen Stereotypen (siehe Tabelle 3.1) erweitert.

Abbildung 3.22 zeigt ein Beispiel für ein aus drei Klassen bestehendes BOTL-Metamodell in der hier definierten UML-Darstellung. Lediglich die Klasse `CClass` verfügt über Tags des Typs `isPK`.

Stereotyp	UML Basisklasse	Beschreibung
`Source Model Variable`	`Package`	Der Stereotyp `Source Model Variable` eines Pakets zeigt an, dass das in dem Paket enthaltene Objektmodell eine Modellvariable gemäß Definition 3.3.8 enthält, welche die linke Seite einer BOTL-Regel darstellt. Das so gekennzeichnete Paket darf ausschließlich Objektmodelle enthalten, deren Objekte wiederum mit Termen versehen sein können (siehe Tabelle 3.2). Falls die zwei Modellvariablen einer Regel durch eine Pfeil (sie Stereotyp `Rule`) unterscheidbar sind, kann die Darstellung des Pakets entfallen, um die Übersichtlichkeit der Darstellung zu erhöhen.
`Target Model Variable`	`Package`	Der Stereotyp `Target Model Variable` legt fest, dass das in dem so gekennzeichneten Paket enthaltenen Objektmodell eine Zielmodellvariable einer BOTL-Regel repräsentiert (siehe Definition 3.3.8). Ebenso wie bei Pakten mit Quellmodellvariablen darf es nur Objektmodelle enthalten, deren Objekte bzw. Links wiederum Objektvariablen bzw. Objektvariablenassoziationen repräsentieren. Bei der Verwendung eines die Modellvariablen einer Regel trennenden Pfeils kann die Darstellung des Pakets entfallen.
`Rule`	`Package`	Wird ein Paket durch diesen Stereotypen gekennzeichnet, so repräsentiert sein Inhalt eine BOTL-Transformationsregel gemäß Definition 3.3.9. Das Paket muss genau ein Paket mit dem Stereotypen `Source Model Variable` und ein Paket mit dem Stereotypen `Target Model Variable` enthalten. Als alternative Darstellung kann der Inhalt dieser beiden Pakete ohne das sie umgebende Paketsymbol angezeigt werden. Die beiden Modellvariablen werden dann optisch durch einen, von der Quellmodellvariablen zur Zielmodellvariablen führenden Pfeil getrennt. Anhand ihrer Position zum Pfeil müssen die beiden Objektvariablen eindeutig unterscheidbar sein.
`Rule Set`	`Package`	Ein mit dem Stereotyp `Rule Set` versehenes UML-Paket repräsentiert ein BOTL-Regelwerk gemäß Definition 3.3.10. Das Paket darf ausschließlich Pakte, die über einen Stereotyp `Rule` verfügen, enthalten. Diese können auch in der alternativen Darstellung mit einem Pfeilsymbol vorliegen. Als alternative Darstellungsform kann das Paketsymbol auch nicht angezeigt werden, falls aus dem Kontext der Anwendung heraus klar ist, dass es sich bei der gegebenen Darstellung um ein BOTL-Regelwerk handelt.

Tabelle 3.1: Die UML-Stereotyen des BOTL-Profils

Tag Definition	Tagged Element	Tag Element Typ	Beschreibung
`isPK`	`Attribute`	`Boolean`	Dieser Tag zeigt an, dass das Attribut ein Primärschlüssel für die Klasse zu der es gehört ist. Falls eine Klasse eine Anzahl von Primärschlüsselattributen hat, so impliziert dies, dass keine zwei Instanzen der Klasse paarweise gleiche Attributwerte in allen diesen Attributen stehen haben darf. Falls dieser Tag an einem Attribut nicht vorhanden ist, wird angenommen, dass das Attribut kein Primärschlüsselattribut ist. Alternativ können Attribute mit dem Tag `isPK` in **Fettschrift** dargestellt werden.
`Term`	`Object`, `DataValue`	`Exp.`	Ein Term als Tagged Value eines `DataValue` legt den Wert des Terms eines Objektvariablenidentifikators oder einer Attributvariablen, zu der der `DataValue` gehört, fest. Gehört der Tagged Value zu einem Attribut, so entspricht dessen Wert im BOTL-Formalismus dem Term *t* einer Objektvariablen aus Definition 3.3.4. Wird `Term` als Tagged Value für ein Element des Typs Objekt selbst verwendet, so ist sein Wert als der Objektvariablenidentifikator *oiv* aus Definition 3.3.4 zu interpretieren. Verfügt die Klasse eines Objektes über Primärschlüssel, so ist für die Objektvariable entweder kein Tagged Value vom Typ `Term` notwendig oder aber sein Wert besteht aus einem Tupel aus den Namen der Primärschlüsselattribute der Klasse. Alternativ können die Primärschlüsselattribute in der Objektvariable auch in **Fettschrift** dargestellt werden. Ein Beispiel für diese Art der Darstellung wurde bereits in Abbildung 3.16 in Abschnitt 3.3 gegeben. Der Wert $\diamondsuit$ kann innerhalb eines Terms auch als $\sharp$ dargestellt werden. Für Attribute und Objekte ohne Primärschlüssel von Objektvariablen, die über keinen Tagged Value des Typs `Term` verfügen, wird der Wert $\diamondsuit$ als Wert angenommen. Als alternative Darstellung ist es weiterhin möglich, den Wert von `Term` direkt anstelle des Datenwerts des UML-Objektes zu schreiben.
`Meta-model`	`Source Model Variable`	`String`	Der Wert dieses Tags bezeichnet das Metamodell einer Quellmodellvariablen.

Tabelle 3.2: Tagged Values des UML-Profils für BOTL

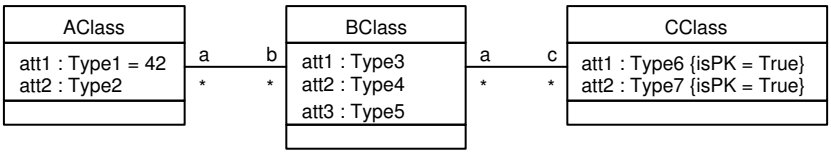


Abbildung 3.22: Beispiel für die Darstellung eines BOTL-Metamodells mit dem UML-Profil für BOTL

Folglich muss für alle Instanzen der Klasse `CClass` (also deren Objekte) sichergestellt werden, dass sie niemals identische Werte in ihren Schlüsselattributen `CClass.att1` *und* `CClass.att2` stehen haben. Das Attribut `att1` der Klasse `AClass` hat einen Initialwert von 42 der innerhalb des BOTL-Formalismus intuitiv als Default-Value interpretiert werden kann.

Für die Darstellung von BOTL-Modellvariablen wird ein UML-Profil für Objektdiagramme verwendet. Objektdiagramme werden durch zwei Tagged Values und einen Stereotypen, die den Tabellen 3.2 und 3.1 entnommen werden können, erweitert.

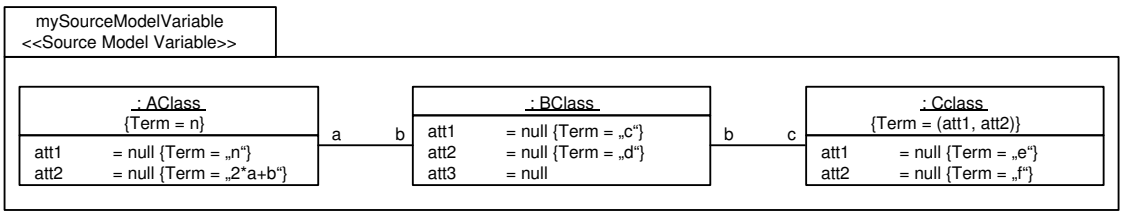


Abbildung 3.23: Beispiel für die Darstellung einer BOTL-Modellvariable mit dem UML-Profil für BOTL

Abbildung 3.23 zeigt eine Objektvariable zu dem in Abbildung 3.22 dargestellten Metamodell. Für jedes Attribut außer dem Attribut `BClass.att3` existiert ein Term. Somit wird implizit angenommen, dass dieses Attribut den Term $\Diamond$ enthält.

Die Objektvariable des Typs `AClass` hat einen Term-Tag mit dem Wert *n* als Identifikator. Dies bedeutet, dass die Identität aller Objekte, die von dieser Objektvariablen erzeugt werden vom jeweils errechneten Wert der Variable *n* abhängen. Im hier vorliegenden Fall taucht diese Variable auch als Term des Attributs `AClass.att1` auf.

Der Identifikator der Objektvariablen des Typs `BClass` hat keinen Term-Tag. Folglich wird der Term $\Diamond$ als Identifikatorterm angenommen. Das bedeutet, dass alle Objekte, die von dieser Objektvariablen erzeugt werden, einen eigenen eindeutigen Identifikator erhalten.

Die beiden Attribute `att1` und `att2` der Klasse `CClass` sind Primärschlüsselattribute. Der Term für den Objektvariablenidentifikator besteht daher aus einem Tupel aus den beiden Namen der Primärschlüsselattribute, wie in Tabelle 3.2 beschrieben.

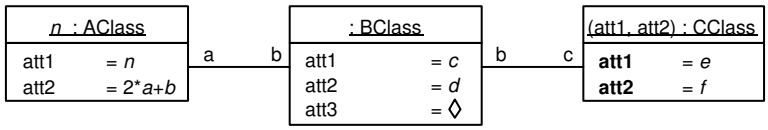


Abbildung 3.24: Eine alternative Darstellungsform der Modellvariable aus Abbildung Abbildung 3.23

Abbildung 3.24 zeigt die gleiche Modellvariable wie 3.23, jedoch in der alternativen graphischen Darstellung. Da diese Form der Darstellung wesentlich intuitiver ist, wird sie im weiteren Verlauf

bevorzugt verwendet werden.

Um Modellvariablen, Regeln und Regelwerke als solche kenntlich zu machen, lassen sich diese in mit entsprechenden Stereotypen versehenen Paketen organisieren. Tabelle 3.1 listet die hierfür vorgesehenen Stereotypen auf. Die Verwendung der hier vorgestellten Tagged Values und Stereotypen ist notwendig, falls BOTL Regelwerke mit UML-Werkzeugen spezifiziert werden sollen. Innerhalb dieser Arbeit wird jedoch die leichter lesbare „Pfeilnotation“ verwendet. Das in Abschnitt 6 vorgestellte Werkzeug zur Spezifikation von BOTL-Regelwerken unterstützt ebenfalls diese Darstellungsweise.

Durch das hier definierte UML-Profil wurde lediglich die konkrete Syntax von BOTL-Regelwerken festgelegt. Die abstrakte Syntax ist durch das UML-Metamodell unter Berücksichtigung der hier eingeführten Stereotypen und Tags gegeben. Wird für BOTL-Regelwerke und Metamodelle ein objektorientiertes Metamodell, ähnlich dem in Abbildung 6.3 (Seite 236) dargestelltem, eingeführt, so lässt sich die Abbildung von UML-Diagrammen auf BOTL selbst wieder durch ein BOTL Regelwerk spezifizieren. Aus Platzgründen wird an dieser Stelle hierauf verzichtet, eine entsprechende Abbildung kann jedoch [BM03a] entnommen werden.

3.5 Regelwerksanwendung

In diesem Abschnitt wird die Semantik von BOTL-Regelwerken definiert, indem formal spezifiziert wird wie diese zur Transformation von Modellen angewendet werden.

Die Anwendung einer BOTL-Regel gliedert sich im Wesentlichen in die folgenden Schritte:

- Im Quellmodell der Regel wird nach einem Fragment gesucht, dessen Struktur dem der Quellmodellvariable entspricht (siehe Def. 3.5.2 zum *Modellfragment-Match*, S. 83). Durch die Terme in den Attributen und Identifikatoren der Quellmodellvariablen können ggf. zusätzliche Constraints über den entsprechenden Werten eines gefunden Modellfragments existieren.
- Ein neues Zielmodellfragment wird entsprechend der Struktur der Zielmodellvariable der Regel erzeugt. Die Attributwerte und Identifikatoren der Objekte dieses neuen Fragments werden aus den Attributwerten und Identifikatoren der Objekte im gefundenen Fragment errechnet (siehe Def. 3.5.4 zur *Modellfragmentrelation* (S. 86) und Def. 3.5.5 zur *Modellfragmenttransformation* (S. 89)).
- Das neu erzeugte Modellfragment wird in das Zielmodellfragment eingefügt. Der spezielle Wert $\diamondsuit$ in einem Attribut kann dabei durch jeden anderen Wert überschrieben werden. Ansonsten führt jeder Versuch einen existierenden Attributwert durch einen anderen zu überschreiben, zu einem ungültigen Ergebnis (siehe Def. 3.5.12 (S. 95) zum *Zusammenführen von Modellfragmenten*).
- Die Regel wird für jeden gefundenen Match der Quellmodellvariable im Quellmodell genau einmal angewendet (siehe Def. 3.5.14 (S. 98) zur *Regelanwendung*).

Bei der Anwendung eines Regelwerks wird jede Regel unabhängig voneinander angewendet und das Ergebnis der Regelanwendungen in einem Zielmodell zusammengeführt (siehe Def. 3.5.16 (S. 99) zur *Regelwerksanwendung*). Die einzelnen Schritte werden im weiteren formal definiert.

3.5.1 Auffinden von Matches in Quellmodellen

Zunächst wird der Begriff *Modellfragment* formal eingeführt. Modellfragmente werden im Laufe der Transformation erzeugt und zu einem Zielmodellfragment zusammengeführt.

Ein Modellfragment ist ähnlich wie ein Modell (s. Def. 3.1.14) ein Geflecht aus Objekten die über Objektassoziationen verbunden sind. Modellfragmente müssen jedoch nicht notwendigerweise konsistent zu einem Metamodell sein, was die Kardinalitäten der enthaltenen Assoziationen angeht. Außerdem dürfen Modellfragmente im Gegensatz zu Modellen $\diamondsuit$ als Attributwerte enthalten.

Definition 3.5.1 (Modellfragment *mf*)
Ein *Modellfragment mf* ist ein Tupel (OB, OA), bestehend aus

- einer Objektbelegung $OB \in \mathbb{OB}$ und
- einer Menge von Objektassoziationen $OA \in \mathscr{P}(\mathbb{OA})$, für die gilt:

$$\forall oa \in OA, oae \in oa|_{OAE} : oae|_o \in OB \tag{3.20}$$

Die Relation *consistent* gilt, falls ein Modellfragment konsistent bezüglich eines gegebenen Metamodells ist:

$$\begin{aligned} consistent(mf, mm) \;:\Leftrightarrow\; & consistent(mf|_{OB}, mm) \;\wedge \\ & \forall oa \in mf|_{OA} : consistent(oa, OB) \end{aligned}$$

$\mathbb{MF}_{mm}$ bezeichnet die Menge aller möglichen Modellfragmente, die zu einem gegebenen Metamodell *mm* konsistent ist. Weiter sei:

$$\mathbb{MF} := \bigcup_{mm \in \mathbb{MM}} \mathbb{MF}_{mm}$$ ❍

Aussage (3.20) entspricht Ausage (3.16) aus der Definition 3.1.14 von Modellen: Die Objektassoziationen eines Modellfragments verbinden nur Objekte, die in diesem Fragment enthalten sind. Hierbei ist durch die Definition von Objektassoziationen bereits sichergestellt, dass Objektassoziationen immer Objekte korrekten Typs verbinden. Allerdings enthält ein Modellfragment nicht notwendigerweise die vom Metamodell der Objektbelegung vorgesehene, korrekte Anzahl von Objektassoziationen.

Modellfragmente sind entweder Teile des Quellmodells, die von einer Regel als passend zur Quellmodellvariablen gefunden werden, oder aber die Fragmente die im Verlauf der Anwendung einer Regel erzeugt werden. Um Modellfragmente mit Modellvariablen in Bezug setzen zu können, wird der *Modellfragment-Match*, oder kurz *Match*, eingeführt. Ein Modellfragment-Match legt die eindeutige Zuordnung von Elementen einer Modellvariable zu Elementen eines Modellfragments fest.

Definition 3.5.2 (Modellfragment-Match *mfm*)
Ein *Modellfragment-Match mfm* ist ein Tupel $(mv, mf, match_{\mathrm{o}}, match_{\mathrm{a}})$, bestehend aus

- einer Modellvariable $mv \in \mathbb{MV}$,
- einem Modellfragment $mf \in \mathbb{MF}$ mit $consistent(mf, mv|_{mm})$,
- einer bijektiven Abbildung

  $$match_{\mathrm{o}} : mv|_{OVB} \to mf|_{OB}$$

  die Objektvariablen auf Objekte passenden Typs abbildet:

  $$\forall ov \in mv|_{OVB} : match_{\mathrm{o}}(ov)|_{ot} = ov|_{otv} \tag{3.21}$$

- einer bijektiven Abbildung

$$match_{\mathrm{a}} : mv|_{OVA} \rightarrow mf|_{OA}$$

die Objektvariablenassoziationen auf Objektassoziationen passenden Typs abbildet:

$$\begin{aligned} &\forall ova \in OVA : \\ &\quad \big(match_{\mathrm{a}}(ova)|_{card} = ova|_{cardv} \wedge \\ &\quad\ match_{\mathrm{a}}(ova)|_{OAE} = \big\{(ovae|_{ae}, match_{\mathrm{o}}(ovae|_{ov})) : ovae \in ova|_{OVAE}\big\}\big) \end{aligned} \tag{3.22}$$

Die Menge aller Modellfragment-Matches wird mit $\mathbb{MFM}$ bezeichnet. ❍

Modellfragment-Matches werden verwendet, um im Quellmodell Modellfragmente zu identifizieren, die zu einer linken Regelseite einer BOTL-Regel passen. Sie werden weiterhin verwendet, um Modellfragmente für das Zielmodell zu konstruieren.

Gemäß Aussage (3.21) liefert $match_{\mathrm{o}}$ für jede Objektvariable ein Objekt, so dass jede Objektvariable der Modellvariable bijektiv auf ein Objekt gleichen Typs des Modellfragments abgebildet wird. $match_{\mathrm{a}}$ liefert zu einer Objektvariablenassoziation eine Objektassoziation, so dass jede Objektvariablenassoziation der Modellvariable bijektiv auf eine Objektassoziation gleichen Typs des Modellfragments abgebildet wird (Aussage (3.22)). Hierbei stimmen auch die Assoziationsenden und die Kardinalitäten der jeweiligen Objektvariablenassoziation und der Objektassoziation überein. Darüber hinaus muss für das Ergebnis von $match_{\mathrm{o}}$ und $match_{\mathrm{a}}$ gelten, dass wenn eine Objektvariable mit einer Objektvariablenassoziation verbunden ist, auch ihre Matches verbunden sind.

Wird in einem Quellmodell nach Fragmenten mit der Struktur der Quellmodellvariablen gesucht, so führt eine solche Suche zu einer Menge gefundener Matches. Im folgenden wird diese *Modellfragment-Match-Menge* formal definiert:

Definition 3.5.3 (Modellfragment-Match-Menge $MFM(mf, mv)$)
Für ein Modellfragment $mf \in \mathbb{MF}$ und eine Modellvariable $mv \in \mathbb{MV}$ mit $consistent(mf, mv|_{mm})$ liefert $MFM(mf, mv) \in \mathscr{P}(\mathbb{MFM})$ die *Modellfragment-Match-Menge* aus Modellfragment-Matches, für die gilt:

1. $\forall mfm_i \in MFM(mf, mv)$:
   - (i) $mfm_i|_{mv} = mv \wedge$
   - (ii) $\forall ov \in mv|_{OVB} : mfm_i|_{match_{\mathrm{o}}}(ov) \in mf|_{OB} \wedge$
   - (iii) $\forall av \in mv|_{OVA} : mfm_i|_{match_{\mathrm{a}}}(av) \in mf|_{OA}$
2. $\forall mfm \in \mathbb{MFM} : mfm$ erfüllt (i) bis $(iii) \Rightarrow mfm \in MFM(mf, mv)$ ❍

In (1.) wird festgelegt, dass jeder Match der Menge ein Match der Modellvariable mv in dem Modellfragment mf ist. D.h. alle Objekte und Assoziationen, die durch die Abbildungen $match_{\mathrm{o}}$ und $match_{\mathrm{a}}$ geliefert werden, müssen aus dem jeweiligen Modellfragment mf stammen. Aussage (2.) stellt schließlich sicher, dass $MFM(mf, mv)$ *alle* möglichen Matches von mv in mf enthält.

Offensichtlich ist die Zahl der Modellfragment-Matches einer gegebenen Modellvariable in einem gegebenen endlichen Modell(fragment) endlich. Für die Anwendung eines Regelwerks wird im Weiteren die Menge aller Modellfragment-Matches in den Quellmodellen benötigt werden.

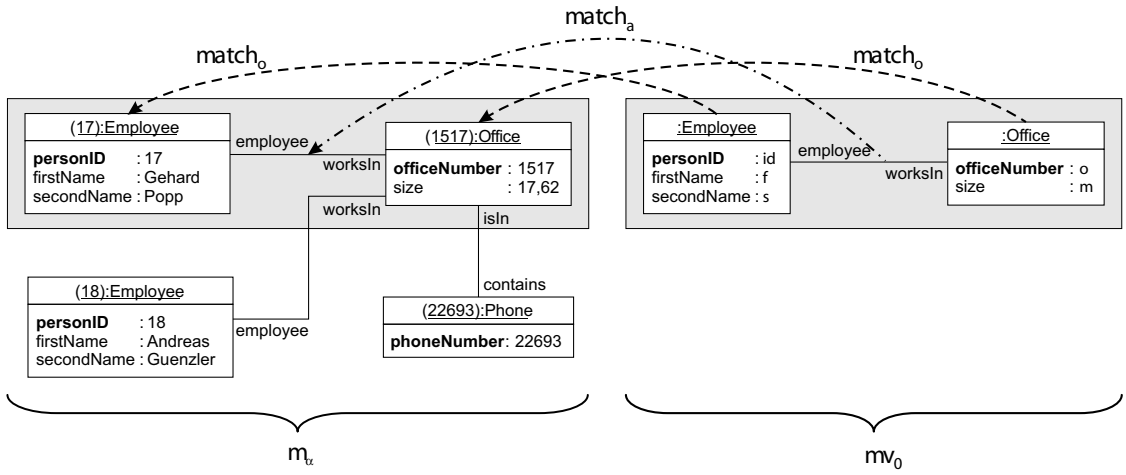


Abbildung 3.25: Ein Modellfragment-Match $(mv_0, mf, match_o, match_a)$

Beispiel: Abbildung Abbildung 3.25 stellt einen der insgesamt zwei möglichen Modellfragment-Matches zwischen dem Model *m* auf der linken Seite und der Modellvariablen $r_0|mv_0$ aus Abbildung 3.17 dar. Jeder Objektvariablen wird genau ein Objekt im Modell auf der linken Seite und jeder Objektvariablenassoziation eine Objektassoziation im Modell zugeordnet.

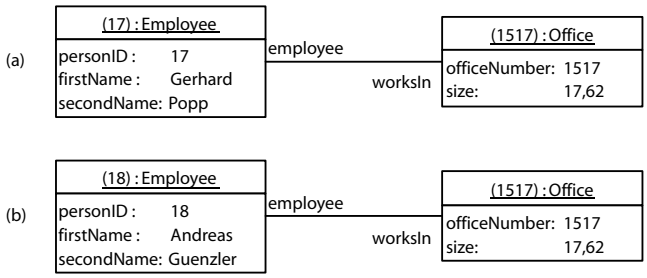


Abbildung 3.26: Die Modellfragmente (a) $mfm_{0,0}|_{mf}$ und (b) $mfm_{0,1}|_{mf}$

Für das Beispiel besteht die Modellfragment-Match-Menge

$$MFM(m_\alpha, r_0|mv_0) = \{mfm_{0,0}, mfm_{0,1}\}$$

aus zwei möglichen Modellfragment-Matches der Modellvariable $r_0|_{mv_0}$ in m_α. Abbildung 3.26 zeigt die Modellfragmente der beiden Modellfragment-Matches $mfm_{0,0}$ und $mfm_{0,1}$. ❍

3.5.2 Transformation von Quellmodellfragmenten in Zielmodellfragmente

Im Folgenden gilt es aus einem Modellfragment-Match der linken Seite einer Regel ein Modellfragment entsprechend der rechten Regelseite zu erzeugen. Die Struktur des erzeugten Modellfragments ist durch die Struktur der Zielmodellvariable der Regel festgelegt, die Attributwerte und Identifikatoren lassen sich anhand der entsprechenden Werte im Quellmodellfragment und den Termen in den beiden Modellvariablen der Regel berechnen. Hierzu wird die Relation *Modellfragmentrelation* mfr

eingeführt. Diese gibt an, ob für einen gegebenen Match zu einer Regel ein Modellfragment die Constraints (in Form eines Gleichungssystems) erfüllt, die durch die Terme in den Modellvariablen formuliert werden. D.h. es beantwortet die Frage: Könnte das Modellfragment für den gegebenen Match von der Regel erzeugt worden sein?

Definition 3.5.4 (Modellfragmentrelation $mfr(mfm, r, mf)$)
Es sei $r \in \mathbb{R}_{mm_0,mm_1}$ eine beliebige Regel mit dem Quellmodell mm_0 und dem Zielmetamodell mm_1, $mfm_0 \in \mathbb{MFM}$ ein Modellfragment-Match und $mf_1 \in \mathbb{MF}$ ein Modellfragment. Eine *Modellfragmentrelation mfr* zwischen diesen Elementen ist dann wie folgt definiert:

$$mfr : \mathbb{MFM} \times \mathbb{R} \times \mathbb{MF} \to \mathbb{B}$$

$$mfr(mfm_0, r, mf_1) :\Leftrightarrow$$

$$mfm_0 \in \bigcup_{mf_0 \in \mathbb{MF}_{mm_0}} MFM(mf_0, r|_{mv_0}) \wedge \tag{3.23}$$

$$\exists mfm_1 \in \mathbb{MFM} : \big(mfm|_{mf} = mf_1 \wedge mfm|_{mv} = r|_{mv_1} \wedge$$
$$\text{Die Objektidentifikatoren und Attribute von } mf_1|_{OB} \text{ bilden} \tag{3.24}$$
$$\text{eine Lösung für das Gleichungssystem } GL.\big)$$

wobei sich das Gleichungssystem GL aus aus vier Teilgeleichungssystemen zusammensetzt.

$$GL := GL^0_{id} \wedge GL^0_v \wedge GL^1_{id} \wedge GL^1_v$$

Diese Gleichungssysteme sind folgendermaßen definiert:

$$GL^0_{id} := \bigwedge_{\substack{ov \in r|_{mv_0}|_{OVB} \\ \wedge\, ov|_{oiv} \neq \Diamond \,\wedge\, ov|_{oiv} \neq \varepsilon \,\wedge\, \neg isTuple(ov|_{oiv})}} mfm_0|_{match_o}(ov)|_{oi} = ov|_{oiv}$$

$$GL^0_v := \bigwedge_{ov \in r|_{mv_0}|_{OVB}} \bigwedge_{att \in ov|_{VV}|_a \wedge ov.att \neq \Diamond} mfm_0|_{match_o}(ov).att = ov.att$$

$$GL^1_{id} := \bigwedge_{ov \in r|_{mv_1}|_{OVB}} \begin{cases} ov|_{oiv} & \text{falls } \varepsilon \neq ov|_{oiv} \neq \Diamond \\ genID(r, mfm_0, ov) & \text{falls } ov|_{oiv} = \Diamond \\ pK(\{(a,v) \in mfm_1|_{match_o(ov)}|_V : \\ \qquad \exists a \in ov|_{otv}|_{Keys}|_n\}) & \text{falls } ov|_{oiv} = \varepsilon \wedge \\ & (\forall (a,t) \in ov|_{VV} \wedge \\ & \quad a \in ov|_{otv}|_{Keys}|_n : t \neq \Diamond) \\ mfm_1|_{match_o}(ov)|_{oi} & \text{sonst} \end{cases}$$
$$= mfm_1|_{match_o}(ov)|_{oi}$$

Hierbei sei

$$genId : \mathbb{R} \times \mathbb{MFM} \times \mathbb{OV} \to \mathbb{ID}$$

eine injektive Funktionen, die für eine Regel, einen Modellfragment-Match und eine Objektvariable einen universell eindeutigen Identifikator erzeugt, der durch keinen anderen Term erzeugt werden kann.

$$GL^1_v := \bigwedge_{ov \in r|_{mv_1}|_{OVB}} \bigwedge_{att \in ov|_{VV}|_a} ov.att = mfm_1|_{match_o}(ov).att$$

❍

Aussage (3.23) impliziert, dass die Modellvariable des Matches mfm_0 dieselbe wie die Quellmodellvariable der Regel r ist, d.h. $mfm_0|mv = r|mv_0$. Der Match mfm_0 matcht in einem beliebigen, zum Quellmetamodell mm_0 konformen Modellfragment.

Aussage (3.24) besagt, dass es einen Match der Zielmodellvariable von r geben muss, der genau auf das Modellfragment mf_1 matcht. Dies impliziert bereits, dass das Modellfragment mf_1 konsistent zum Zielmetamodell mm_1 der Regel sein muss (kurz:$consistent(mf_1, mm_1)$). Existiert ein solcher Match, so müssen die im Modellfragment mf_1 enthaltenen Attribut- und Identifikatorwerte Lösungen des Gleichungssystems GL sein, welches zur Berechnung dieser Werte für ein erzeugtes Zielmodellfragment verwendet wird.

GL^0_{id} ist die Menge von Gleichungen, die Objektidentifikatoren eines Quellmodellfragments zu Objektvariablenidentifikatoren einer linken Regelseite in Bezug setzen. Objektvariablen mit $\diamondsuit$, Primärschlüsselattributen oder Tupeln als Identifikator werden nicht im Gleichungssystem berücksichtigt. Der Wert $\diamondsuit$ in einer Quellmodellvariablen steht somit für einen beliebigen Wert in einem Quellmodell. Tupel als Identifikatoren werden genauso wie der Wert $\diamondsuit$ behandelt. Der Grund hierfür liegt darin, dass sich die Identität eines gefundenen Quellobjektes in der Praxis nicht ohne weiteres als Tupel darstellen lässt. Die Möglichkeit, dennoch Tupel als Identifikatoren auf der linken Regelseite verwenden zu können, ist daher lediglich für bidirektionale Abbildungen (siehe auch Abschnitt 4.3, S. 167 ff.) von Bedeutung.

GL^0_v ist die Menge der Gleichungen, die Attributwerte eines Quellmodellfragments zu allen Attributwerten der Objektvariablen einer linken Regelseite in Bezug setzen. Gleichungen für Attribute mit einem Wert von $\diamondsuit$ als Term werden auch hier ignoriert.

GL^1_{id} umfasst die Menge aller Gleichungen, die Objektidentifikatoren von Objektvariablen einer rechten Regelseite betreffen. Hierbei werden vier mögliche Fälle unterschieden: Im ersten Fall enthält der Identifikator der Objektvariablen einen Term, der mit dem Identifikator des erzeugten Objektes gleichgesetzt wird. Im zweiten Fall, in dem der Objektvariablenidentifikatorterm den Wert $\diamondsuit$ hat, wird jeweils ein eindeutiger Identifikator für das neue Objekt generiert. Verfügt die Klasse der Objektvariablen über Primärschlüsselattribute (dritter Fall), so wird die Identität des generierten Objektes aus den Werten dieser Attribute errechnet, insofern keines von ihnen den Wert $\diamondsuit$ enthält. Interessant ist vor allem der letzte Fall: Hier enthält mindestens ein Primärschlüsselattribut der Objektvariablen den Wert $\diamondsuit$. Der Identifikator des neu erzeugten Objektes lässt sich nun lediglich anhand einer Gleichung der Form

$$mfm_1|_{match_o}(ov)|_{oi} = mfm_1|_{match_o}(ov)|_{oi}$$

errechnen, d.h. die Lösung ist beliebig bzw. es lässt sich keine eindeutige Lösung ermitteln.

GL^1_v umfasst die Mengen aller Gleichungen, welche die Attributwerte von Objektvariablen einer rechten Regelseite betreffen. Hierbei werden Gleichungen für Attribute mit dem Wert $\diamondsuit$ *nicht* ignoriert. Attribute für die sich dieser Wert ergibt, werden beim Abschluss einer Modelltransformation durch passende Default-Werte ersetzt.

In der nachfolgenden Definition 3.5.5 für Modellfragmenttransformationen ist festgelegt, dass solche Gleichungssysteme zu dem ungültigen Ergebnis $\perp$ führen. Alternativ ließe sich für den vierten Fall analog zum zweiten Fall ein eindeutiger Identifikator generieren. Die Art der hier gemachten Festlegung ist jedoch sinnvoll, da am Ende einer Modelltransformation verbleibende $\diamondsuit$-Werte mit Default-Werten belegt werden (siehe auch Definition 3.5.16, S. 99). Werden für Objekt mit Primärschlüsselattributen Identifikatoren generiert, so kann dies dazu führen, dass nach der Modelltransformation mehrere Objekte mit identischen Primärschlüsseln und unterschiedlichem Identifikator existieren, was zu einem inkonsistenten Zielmodell führen würde.

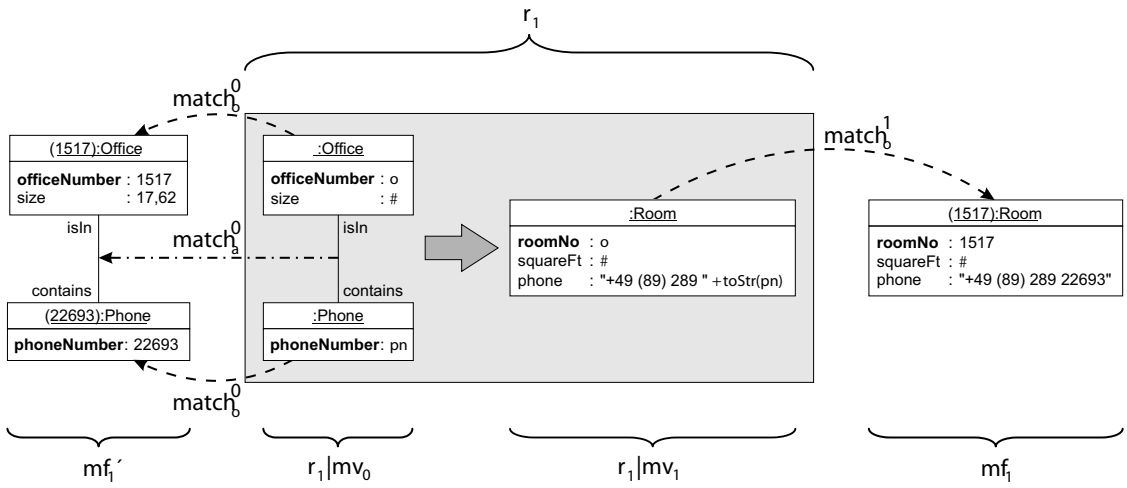


Abbildung 3.27: Die Regel r_1 mit einem Modellfragment-Match auf der linken Seite und einem Modellfragment mf_1, wobei $mfr(mfm_i^0, r_1, mf_1)$ gilt

Beispiel: Abbildung 3.27 stellt eine weitere Regel r_1 dar, die ebenso wie Regel r_0 (siehe Abbildung 3.18) Instanzen des Metamodells mm_α (siehe Abbildung 3.5) in Instanzen des Metamodells mm_β aus Abbildung 3.6 transformiert.

Auf der linken Seite der Regel ist ein gültige Modellfragmentrelation für r_1 abgebildet. Die linke Modellvariable $r_1|_{mv_0}$ der Regel r_1 besteht hierbei aus den beiden Objektvariablen $ov_{0,0}$ vom Typ Office und $ov_{0,1}$ vom Typ Phone sowie einer Objektvariablenassoziation zwischen ihnen. Die Modellvariable $r_1|_{mv_1}$ besteht lediglich aus einer einzigen Objektvariable, die im Folgenden als $ov_{1,0}$ bezeichnet wird. Der einzige Modellfragment-Match $mfm_{1,0} \in MFM(m_\alpha, r_1|_{mv_0})$ zwischen der linken Modellvariable $r_1|_{mv_0}$ und dem Modellfragment mf_1' wird durch gestrichelte Pfeile angedeutet.

Wie in Definition 3.5.4 (i) gefordert, existiert ebenfalls ein Match $mfm_{1,0}^1$ für die rechte Modellvariable $r_1|_{mv_1}$ der ein neues Modellfragment mf_1 erzeugt. Gemäß Definition 3.5.4 ergibt sich für GL das folgende Gleichungssystem:

$$
\begin{aligned}
GL_{id}^0 : (\quad & match_o^0(ov_{0,0})|oi && = \varepsilon \quad \text{ignoriert gemäß Def. 3.5.4)} \\
(\quad & match_o^0(ov_{0,1})|oi && = \varepsilon \quad \text{ignoriert gemäß Def. 3.5.4)} \\
GL_v^0 : \quad & match_o^0(ov_{0,0}).office Number && = o \\
& match_o^0(ov_{0,0}).size && = m \\
& match_o^0(ov_{0,1}).phoneNumber && = pn \\
GL_{id}^1 : \; & pK(\{(roomNo, match_o^1(ov_{1,0}).roomNo)\}) && = match_o^1(ov_{1,0})|oi \\
GL_v^1 : \quad & o && = match_o^1(ov_{1,0}).roomNo \\
& \diamondsuit && = match_o^1(ov_{1,0}).squareFt \\
& \text{+49 (89) 289} \circ \text{toStr}(pn) && = match_o^1(ov_{1,0}).phone
\end{aligned}
$$

Für das Gleichungssystem existiert eine Lösung, wobei die Identifikatoren und Attributwerte der erzeugten Objekte im Zielmodellfragment als die Unbekannten des Gleichungssystems angesehen wer-

den:

$$\begin{aligned} match^1_o(ov_{1,0})|oi &= pK(\{(roomNo, match^0_o(ov_{0,0}).officeNumber\}) \\ match^1_o(ov_{1,0}).roomNo &= match^0_o(ov_{0,0}).officeNumber \\ match^1_o(ov_{1,0}).squareFt &= \diamondsuit \\ match^1_o(ov_{1,0}).phone &= \text{“+49 (89) 289 ”} \circ \text{toStr}(match^0_o(ov_{0,1}).phoneNumber) \end{aligned}$$

Für den dargestellten Match erhält man für das erzeugte Zielmodellfragment folglich die folgenden Werte:

$$\begin{aligned} match^1_o(ov_{1,0})|oi &= pK(\{(roomNo, 1517\}) \\ match^1_o(ov_{1,0}).roomNo &= 1517 \\ match^1_o(ov_{1,0}).squareFt &= \diamondsuit \\ match^1_o(ov_{1,0}).phone &= \text{“+49 (89) 289 ”} \circ \text{toStr}(22693) = \text{“+49 (89) 289 22693”} \end{aligned}$$

Demnach hat das Gleichungssystem *GL* eine eindeutige Lösung für alle unbekannten Variablen und die Relation $mfr(mfm^0_i, r_1, mf_1)$ gilt. ❍

Generell lassen sich nicht alle Gleichungssysteme so leicht wie in diesem Beispiel lösen. So wird hier der Konkatenationsoperator “$\circ$” für Strings verwendet. Da dieser Operator zusammen mit der Menge der Strings keine mathematische Gruppe bildet, müssen für eine Werkzeugunterstützung geeignete Strategien entwickelt werden, wie mit solchen Operatoren umgegangen wird.

Die in Definition 3.5.4 eingeführte Modellfragmentrelation sagt aus, ob ein Zielmodellfragment, die durch die in den Termen und Attributen formulierten Constraints erfüllt, um eine gültige Lösung einer Anwendung einer Regel sein zu können. Interessant sind die Fälle, in denen es nur maximal eine solche Lösung geben kann, da in solch einem Fall das durch die Anwendung der Regel erzeugte Zielfragment deterministisch berechenbar ist. Im Folgenden wird eine solche Abbildung als *Modellfragmenttransformation* eingeführt.

Definition 3.5.5 (Modellfragmenttransformation $mft(mfm_0, r)$)
Seien $mm_0, mm_1 \in \mathbb{MM}$ das Quell- bzw. Zielmetamodell einer Regel $r \in \mathbb{R}_{mm_0,mm_1}$, $mf_0 \in \mathbb{MF}_{mm_0}$ ein Modellfragment und $mfm_0 \in MFM(mf_0, r_{mv_0})$ ein Modellfragment-Match.
Eine *Modellfragmenttransformation mft* des Modellfragment-Matches mfm_0 durch die Regel r ist dann eine Abbildung, für die gilt:

$$mft : \mathbb{MFM} \times \mathbb{R} \rightarrow \mathbb{MF} \cup \{\bot\}$$

$$mft(mfm_0, r) \mapsto \begin{cases} mf_1 & \text{falls } \exists_1 mf_1 : mfr(mfm_0, r, mf_1) \ \wedge mf_1 \in \mathbb{MF}|_{mm_1} \\ (\emptyset, \emptyset) \in \mathbb{MF}|_{mm_1} & \text{falls } \nexists\, mf_1 : mfr(mfm_0, r, mf_1) \\ \bot & \text{sonst} \end{cases}$$

❍

Dementsprechend liefert eine Modellfragmenttransformation zu einem Modellfragment-Match der linken Regelseite einer Regel ein neues Modellfragment, falls sich die Werte von dessen Identifikatoren und Attributen gemäß Definition 3.5.4 eindeutig berechnen lassen.

Existiert keine Lösung für die Attribut- und Identifikatorwerte, so liefert *mft* ein leeres Modellfragment. Dies tritt z.B. ein, falls falls die linke Regelseite einen konstanten Wert für ein Attribut erwartet, der jedoch nicht mit dem durch den Match gefundenen übereinstimmt. Ein anderer Fall, in dem ein leeres Fragment erzeugt wird, ergibt sich, wenn die Berechnung eines Wertes für das erzeugte

Modellfragment nicht möglich ist, z.B. falls der Wert eines Attributs sich aus $\sqrt{x}$ errechnet und x im Quellmodellfragment mit einem negativen Wert belegt wird.

Existieren mehrere mögliche Lösungen für einen Wert im erzeugten Zielfragment, so liefert *mft* den Wert $\perp$ als Ergebnis. In diesem Fall ist keine eindeutige Transformation möglich und das Ergebnis wird als ungültig angesehen.

$\perp$ wird auch als Ergebnis geliefert, falls genau ein Modellfragment mf_1 die Relation

$$mfr(mfm_i, r, mf_1)$$

erfüllt aber die zusätzliche Forderung

$$mf_1 \in \mathbb{MF}|_{mm_1}$$

nicht erfüllt ist. In diesem Fall ist es zwar möglich eindeutige Attribut- und Identifikatorwerte für das neu erzeugte Modellfragment zu berechnen, allerdings ist dieses Fragment nicht mehr konsistent zu seinem Metamodell gemäß Definition 3.5.1 (s. S. 83). Als einziger Grund kommt hierfür der Fall in Frage, dass gilt

$$\neg consistent(mf_1|_{OB}, mm_1)$$

Eine inkonsistente Objektbelegung kann hierbei lediglich daraus resultieren, dass zwei Objekte gleichen Typs denselben Identifikator haben.

Die Rückgabe eines leeren Modellfragments bei nicht lösbaren Gleichungssystemen *GL* ermöglicht es, die Auswahl von Quellmodellfragmenten, aus denen neue Fragmente erzeugt werden, nicht nur von deren Struktur, sondern auch auf deren Inhalt (im Wesentlichen deren Attributwerte) abhängig zu machen. Im Folgenden werden Regeln, die auch Attribut- bzw. Identifikatorwerte bei der Auswahl von Quellfragmenten zur Transformation berücksichtigen, als *Constraint-behaftete* Regeln bezeichnet.

Definition 3.5.6 (Constraint-behaftete Regeln (*constrained*(r)))
Eine Regel $r \in \mathbb{R}$ heißt genau dann *Constraint-behaftet*, wenn das nachstehende Prädikat *constrained*(r) gilt:

$$constrained : \mathbb{R} \to \mathbb{B}$$
$$constrained(r) :\Leftrightarrow \exists m \in \mathbb{M}_{r|_{mv_0}|_{mm}}, mfm \in MFM(m, r|_{mv_0}) :$$
$$mft(mfm, r) = (\emptyset, \emptyset)$$ ❍

Das folgende Lemma bietet eine Technik an, anhand derer sich feststellen lässt ob eine gegeben Regel Constraint-behaftet ist oder nicht:

Lemma 3.5.1 (Constraint-behaftete Regeln)
Eine Regel $r \in \mathbb{R}$ ist genau dann Constraint-behaftet, falls das Gleichungssystem *GL* aus Definition 3.5.4 der Regel r für jeden Identifikator- und Attributwert der rechten Regelseite einen eingeschränkten Lösungsraum besitzt. (Beweis s. A.1.2, S. 280) ❑

Beispiel: Es werden nun die Modellfragmenttransformationen für die Matches der Regel r_0 (siehe Abbildung 3.18, S. 75) im Quellmodell m_α (siehe Abbildung 3.10 S. 62) betrachtet. Jede der Modellfragmenttransformationen liefert ein neues Modellfragment dessen Struktur durch die rechte Seite der jeweils angewendeten Regel festgelegt ist.

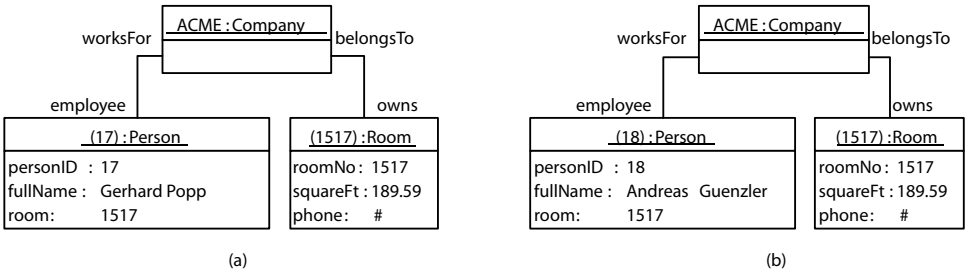


Abbildung 3.28: Die erzeugten Modellfragmente (a) $mf_{0,0}$ und (b) $mf_{0,1}$

Da alle Werte der Modellfragmentrelation im Beispiel deterministisch berechnet werden können, liefert die Modellfragmenttransformation gemäß Definition 3.5.5 immer einen Wert ungleich $\perp$. Das erste erzeugte Modellfragment $mf_{0,0} := mft(mfm_{0,0}, r_0)$ ist in Abbildung 3.28 (a) dargestellt. Das zweite durch die Regelanwendung erzeugte Modellfragment $mf_{0,1} := mft(mfm_{0,1}, r_0)$ ist in Abbildung 3.28 (b) dargestellt. ❍

Aus Objektvariablen, die über Tupel unterschiedlicher Stelligkeit oder unterschiedliche konstante Werte als Identifikatorterme verfügen, lassen sich offensichtlich niemals dieselben Objekte erzeugen, da die erzeugten Identifikatoren immer unterschiedlich sein werden. Die damit verbunden Möglichkeit, Aussagen darüber zu machen, ob zwei Objektvariablen dasselbe Objekt erzeugen können ist insbesondere für die in Abschnitt 4 vorgestellten Verifikationstechniken hilfreich. Aus diesem Grund wird an dieser Stelle ein Ähnlichkeitsbegriff für Identifikatorterme eingeführt.

Definition 3.5.7 (Ähnliche Identifikatorterme $oiv_0 \sim oiv_1$, $isTuple(oiv)$)
Zwei Identifikatorterme $oiv_0, oiv_1 \in Term_{\mathbb{ID}}$ werden genau dann als *ähnlich* bezeichnet, falls das im folgenden definierte Infix-Relation $\sim$ gilt:

$$\begin{aligned}
\sim &: Term_{\mathbb{ID}} \times Term_{\mathbb{ID}} \to \mathbb{B} \\
&oiv_0 \sim oiv_1 :\Leftrightarrow oiv_0, oiv_1 \in \mathbb{ID} \cup \mathbb{VAR} \wedge \neg(\mathbb{ID} \ni oiv_0 \neq oiv_1 \in \mathbb{ID}) \vee \\
&\qquad oiv_0 = (s_0, \ldots, s_n), oiv_1 = (t_0, \ldots, t_n) \wedge \\
&\qquad \nexists i \in \{0, \ldots, n\} : \mathbb{ID} \ni s_i \neq t_i \in \mathbb{ID}
\end{aligned}$$

Weiter gilt für einen Identifikatorterm $oiv \in Term_{\mathbb{ID}}$ das Hilfsprädikat $isTuple(oiv)$ genau dann, wenn oiv ein n-Tupel mit $1 < n \in \mathbb{N}^+$ ist. ❍

Die nachfolgende Definition verwendet den Ähnlichkeitsbegriff für Identifikatorterme und weitet ihn auf Objektvariablen aus. Ziel ist es, für zwei Objektvariablen für die diese Relation *nicht* gilt, nachweisen zu können, dass sie mit Sicherheit keine identischen Objekte erzeugen können.

Definition 3.5.8 (Ähnliche Objektvariablen $ov_0 \sim ov_1$)
Zwei Objektvariablen $ov_0, ov_1 \in \mathbb{OV}$ heißen genau dann *ähnlich*, wenn die im folgenden definierte

Infix-Relation $\sim$, gilt:

$$\sim: \mathbb{OV} \times \mathbb{OV} \to \mathbb{B}$$
$$ov_0 \sim ov_1 :\Leftrightarrow$$
$$(i)\ ov_0|_{otv} = ov_1|_{otv} \wedge$$
$$(ii)\ ov_0|_{otv}|_{Keys} = \emptyset \wedge ov_0|_{oiv} \sim ov_1|_{oiv} \vee$$
$$ov_0|_{otv}|_{Keys} \neq \emptyset \wedge \forall (n,t) \in ov_0|_{otv}|_{Keys} : \left(ov_0.n \notin t|_{T_{val}} \vee ov_1.n \notin t|_{T_{val}} \vee ov_0.n = ov_1.n\right) \ \text{❍}$$

Das folgende Lemma fast nun die gewünschte Aussage zusammen:

Lemma 3.5.2 (Objekte aus ähnlichen Objektvariablen)
Sein $R \in \mathbb{RW}$ ein Regelwerk. Dann können zwei Zielobjektvariable $ov_i, ov_j \in R|_{mv_1}|_{OVB}$ des Regelwerks, für die

$$ov_i \not\sim ov_j$$

gilt, niemals dasselbe Objekt im Zielmodell erzeugen. (Beweis s. A.1.2, S. 280) ❑

Das hier vorgestellte Lemma ist insbesonders für die in Kapitel 4 vorgestellten Verifikationstechniken für die Anwendbarkeit und Metamodellkonformität von Regelwerken von großer Bedeutung.

3.5.3 Zusammenführen der erzeugten Modellfragmente

Eine Modellfragmenttransformation transformiert ein Modellfragment eines Quellmodells in genau ein Modellfragment des Zielmodells. Um ein vollständiges Modell zu transformieren müssen sämtliche gefundenen Matches transformiert und die Ergebnisse der Transformationsschritte zusammengesetzt werden. Hierzu wird im Folgenden die Relation *mergeable* definiert, das angibt ob zwei Objektbelegungen vereinigt werden können. Weiterhin wird ein Operator für das Zusammenführen von Modellfragmenten eingeführt.

Zunächst wird die Relation *attMergeable* eingeführt. Diese Relation wird im Verlauf des Verschmelzens zweier Objekte mit derselben Identität und demselben Typ zu einem Objekt benötigt, um festzustellen, ob die beiden Objekte widersprüchliche Attributwerte beinhalten.

Definition 3.5.9 (*attMergeable*$((a_0,v_0),(a_1,v_1))$)
Es seien $(a_0,v_0),(a_1,v_1) \in \mathbb{ID} \times \mathbb{T}|_{T_{\text{val}}} \cup \{\diamondsuit\}$ zwei Objektattribute Dann ist die Relation *attMergeable* definiert als:

$$attMergeable : (\mathbb{ID} \times \mathbb{T}|_{T_{\text{val}}} \cup \{\diamondsuit\}) \times (\mathbb{ID} \times \mathbb{T}|_{T_{\text{val}}} \cup \{\diamondsuit\}) \to \mathbb{B}$$
$$attMergeable((a_0,v_0),(a_1,v_1)) :\Leftrightarrow a_0 = a_1 \wedge (v_0 = v_1 \vee \diamondsuit \in v_0 \cup v_1) \ \text{❍}$$

Die Relation *mergeable* gilt für zwei Objektbelegungen genau dann, wenn es möglich ist sie zu einer Objektbelegung zu verschmelzen. Insbesonders dürfen die beiden Objektbelegungen keine identischen Objekte mit widersprüchlichen Attributwerten beinhalten.

Definition 3.5.10 (*mergeable*(OB_0, OB_1))
Es seien $OB_0, OB_1 \in \mathbb{OB}$ zwei Objektbelegungen. Dann gilt für die Relation *mergeable*:

$$
\begin{aligned}
&mergeable : \mathbb{OB} \times \mathbb{OB} \to \mathbb{B} \\
&mergeable(OB_0, OB_1) :\Leftrightarrow \nexists o_0 \in OB_0, o_1 \in OB_1 : \\
&\qquad o_0|_{oi} = o_1|_{oi} \wedge o_0|_{ot} = o_1|_{ot} \wedge \\
&\qquad \big(\exists (a_0, v_0) \in o_0|_V, (a_1, v_1) \in o_1|_V : \\
&\qquad\quad a_0 = a_1 \wedge \neg attMergeable((a_0, v_0), (a_1, v_1))\big)
\end{aligned}
$$
❍

Für zwei Objektbelegungen trifft *mergeable* also zu, falls in den beiden Belegungen keine zwei Objekte gleichen Typs mit dem gleichen Identifikator und unterschiedlichen Attributwerten vorkommen. Eine wichtige Eigenschaft dieser Relation, die im nachstehenden Lemma festgehalten wird, ist ihre Kommutativität:

Lemma 3.5.3 (Kommutativität von *mergeable*)
Die Relation *mergeable* ist kommutativ. (Beweis s. A.1.2, S. 281) ❑

Lemma 3.5.4 führt eine Reihe von Eigenschaften der Relation *mergeable* an, die für weitere Beweise benötigt werden:

Lemma 3.5.4 (Eigenschaften von *mergeable*)
Seien $OB_0, OB_1 \in \mathbb{OB}$ Objektbelegungen. Dann gilt für beliebige $OB'_0 \subseteq OB_0, OB'_1 \subseteq OB_1$:

(i) $mergeable(OB_0, OB_1) \Rightarrow mergeable(OB'_0, OB'_1)$

(ii) $\neg mergeable(OB_0, OB_1) \Rightarrow \exists o_0 \in OB_0, o_1 \in OB_1$ mit $\neg mergeable(\{o_0\}, \{o_1\})$

(iii) $\neg mergeable(OB'_0, OB'_1) \Rightarrow \neg mergeable(OB_0, OB_1)$ (Beweis s. A.1.2, S. 282) ❑

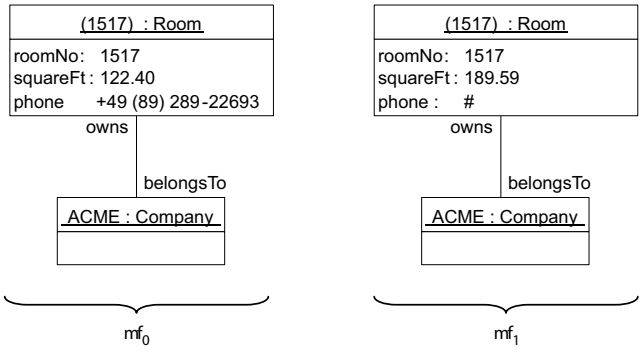


Abbildung 3.29: Die Modellfragmente mf_0 und mf_1

Beispiel: Abbildung 3.29 stellt zwei Modellfragmente mf_0 und mf_1 dar für deren Objektbelegungen gilt:

$$\neg mergeable(mf_0|_{OB}, mf_1|_{OB})$$

Der Grund hierfür ist, dass das Attribut squareFt in den beiden Objekten des Typs Room zwei unterschiedliche Werte enthält, obwohl beide Objekte denselben Primärschlüsselwert im Attribut RoomNo aufweisen. Dementsprechend gilt für diese beiden Objektattribute gemäß Definition 3.5.9:

$$\neg attMergeable((\text{squareFt}, 122, 40), (\text{squareFt}, 189.59))$$

❍

Die nachfolgende Definition legt fest, wie zwei Objektbelegungen zusammengeführt werden. Hierzu wird die Abbildung *OBmerge* eingeführt.

Definition 3.5.11 (*OBmerge*(OB_0, OB_1))
Es sei *OBmerge* eine strikte Abbildung von zwei Objektbelegungen OB_0 und OB_1 auf eine andere Objektbelegung, so dass gilt:

$$OBmerge : \mathbb{OB} \times \mathbb{OB} \to \mathbb{OB}$$

$$OBmerge(OB_0, OB_1) \mapsto \begin{cases} OB^+ & \text{fall gilt: } mergeable(OB_0, OB_1) \\ \bot & \text{sonst} \end{cases}$$

Hierbei ist OB^+ die Menge von Objekten für die gilt:

$$\forall o_0 \in OB_0 \text{ mit } o_0|_{oi} \notin \{o \in OB_1 : o|_{ot} = o_0|_{ot}\}|_{oi} : o_0 \in OB^+$$

$$\forall o_1 \in OB_1 \text{ mit } o_1|_{oi} \notin \{o \in OB_0 : o|_{ot} = o_1|_{ot}\}|_{oi} : o_1 \in OB^+$$

$$\forall o_0 \in OB_0, o_1 \in OB_1 \text{ mit } o_0|_{oi} = o_1|_{oi} =: oi^+ \wedge o_0|_{ot} = o_1|_{ot} =: c^+ :$$

$$\exists_1 o^+ \in OB^+ : o^+|_{oi} = oi^+ \wedge$$

$$o^+|_{ot} = c^+ \wedge$$

$$o^+|_V = \Big\{(a, v) : \quad (a, v_0) \in o_0|_V \wedge (a, v_1) \in o_1|_V$$

$$\wedge v = \begin{cases} v_0 & \text{falls } v_0 \neq \diamondsuit \\ v_1 & \text{sonst} \end{cases} \Big\}$$

❍

Das Ergebnis von *OBmerge* ist eine Objektbelegung, die sämtliche Objekte der beiden Ursprungsbelegungen enthält, falls für diese die Relation *mergeable* gilt. Die Attribute der resultierenden Objekte enthalten dieselben Werte wie in den Ursprungsbelegungen, jedoch können $\diamondsuit$-Werte durch beliebige andere Werte überschrieben werden. Sind zwei Objektbelegungen nicht *mergeable* (z.B. falls es zu Konflikten zwischen Attributwerten kommt), so liefert *OBmerge* den Fehlerwert $\bot$. Da *OBmerge* eine strikte Abbildung ist liefert *OBmerge* immer $\bot$ als Ergebnis, falls eines seiner Argumente gleich $\bot$ ist.

Im Folgenden wird eine Reihe von Eigenschaften der *OBmerge*-Abbildung festgehalten, die für weitere Beweise nützlich sind. Zunächst kann festgestellt werden, dass *OBmerge* eine kommutative Abbildung ist:

Lemma 3.5.5 (Kommutativität von *OBmerge*)
Die strikte Abbildung *OBmerge* ist kommutativ. (Beweis s. A.1.2, S. 282) ❑

Liefert die Abbildung *OBmerge* für zwei Objektbelegungen den Wert $\bot$, so erhält man dasselbe Ergebnis für beliebige Obermengen der beiden Objektbelegungen. Lemma 3.5.6 hält diese Eigenschaft fest:

Lemma 3.5.6 (*OBmerge* von Obermengen von Objektbelegungen)
Seien $OB_0, OB_1 \in \mathbb{OB}$ Objektbelegungen. Für beliebige $OB'_0 \subseteq OB_0, OB'_1 \subseteq OB_1$ gilt:

$$\begin{aligned} & OBmerge(OB'_0, OB'_1) = \bot \\ \Rightarrow\ & OBmerge(OB_0, OB_1) = \bot \end{aligned}$$

(Beweis s. A.1.2, S. 282) ❑

Das nachstehende Lemma besagt, dass für einen beliebigen Typ keine Objektidentifikatoren von Objekten dieses Typs durch die Anwendung der *OBmerge*-Operation verloren gehen:

Lemma 3.5.7 (*OBmerge* erhält Objektidentifikatoren)
Seien $OB_0, OB_1 \in \mathbb{OB}$ Objektbelegungen. Dann gilt:

$$\begin{aligned} & \forall c \in \mathbb{C} : \\ & \quad o|_{oi} \in \{o \in OB_0 : o|_{ot} = c\}|_{oi} \cup \{o \in OB_1 : o|_{ot} = c\}|_{oi} \ \wedge mergeable(OB_0, OB_1) \\ \Leftrightarrow\ & o|_{oi} \in \{o \in OBmerge(OB_0, OB_1) : o|_{ot} = c\}|_{oi} \end{aligned}$$

(Beweis s. A.1.2, S. 282) ❑

Lemma 3.5.8 (Eigenschaften von *mergeable*)
Seien $OB_0, OB_1, OB_2 \in \mathbb{OB}$ Objektbelegungen. Falls $OBmerge(OB_0, OBmerge(OB_1, OB_2)) \neq \bot$ gilt, so gilt auch:

(i) $mergeable(OB_0, OB_1)$

(ii) $mergeable(OB_0, OB_2)$

(iii) $mergeable(OB_1, OB_2)$

(Beweis s. A.1.2, S. 282) ❑

Mit Hilfe der bereits vorgestellten Eigenschaften der *OBmerge*-Abbildung lässt sich nun auch die Assoziativität dieser Abbildung nachweisen. Lemma 3.5.9 hält diese Eigenschaft fest:

Lemma 3.5.9 (Assoziativität von *OBmerge*)
Die Abbildung *OBmerge* ist assoziativ.

(Beweis s. A.1.2, S. 283) ❑

Aufbauend auf der Definition zum Zusammenführen von Objektbelegungen wird nun die Operation $\cup_{\mathfrak{m}}$ zum Zusammenführen von Modellfragmenten eingeführt. Diese wird benötigt, um die im Verlauf einer Modelltransformation erzeugten Zielmodellfragmente in das zu erzeugende Zielmodell zu integrieren.

Definition 3.5.12 (Merge $mf_0 \cup_{\mathfrak{m}} mf_1$)
$\cup_{\mathfrak{m}}$ ist eine strikte Abbildung:

$$\cup_{\mathfrak{m}} : \mathbb{MF} \times \mathbb{MF} \to \mathbb{MF} \cup \{\bot\}$$

$$mf_0 \cup_{\mathfrak{m}} mf_1 \mapsto \begin{cases} \bot & \text{für } mf_i = \bot \text{ mit } i \in \{0,1\} \\ & \vee \neg mergeable(mf_0|_{OB}, mf_1|_{OB}) \\ (OBmerge(mf_0|_{OB}, & \\ \quad mf_1|_{OB}), OA^+) & \text{sonst} \end{cases}$$

Mit

$$OA^+ = (mf_0|_{OA} \cup mf_1|_{OA}) \setminus \{\, (OAT, card, OAE) : \exists oa_k \in mf_k \text{ mit } k \in \{0,1\} : \\ OAT = oa_k|_{OAT} \\ OAE = oa_k|_{OAE} \\ oa_0|_{card} \neq oa_1|_{card} \\ card = \min\{oa_0|_{card}, oa_1|_{card}\} \,\}$$

Sei $\{mf_0, \ldots, mf_n\} \in \mathscr{P}(\mathbb{MF})$ eine Menge von Modellfragmenten. Dann gilt die folgende Kurzschreibweise:

$$\bigcup_{\mathfrak{m}\ mf_i \in \{mf_0,\ldots,mf_n\}} mf_i := mf_0 \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} mf_n$$ ❍

Das Zusammenführen von $\perp$ mit beliebigen anderen Fragmenten liefert immer $\perp$. Zwei Objekte mit demselben Identifikator werden zu einem Objekt zusammengeführt. Die zusammengehörigen Attribute müssen hierbei denselben Wert haben oder zumindest eines von ihnen muss den Wert $\diamondsuit$ haben. Im letzten Fall wird der $\diamondsuit$-Wert durch den jeweils anderen überschrieben. Ist dies nicht gegeben, so wird $\perp$ als Ergebnis geliefert. Sämtliche Assoziationen zwischen Objekten bleiben erhalten. OA^+ ist prinzipiell die Vereinigungsmenge aller Objektassoziationen der Modellfragmente mf_0 und mf_1, jedoch werden bei gleichen Assoziationen zwischen zwei Objekten in mf_0 und mf_1 die mit der kleineren Kardinalität nicht in die Menge OA^+ aufgenommen. Folglich ist die Kardinalität einer erhaltenen Objektassoziation immer gleich dem Maximum der Kardinalitäten der Assoziationen in den Ursprungsfragmenten.

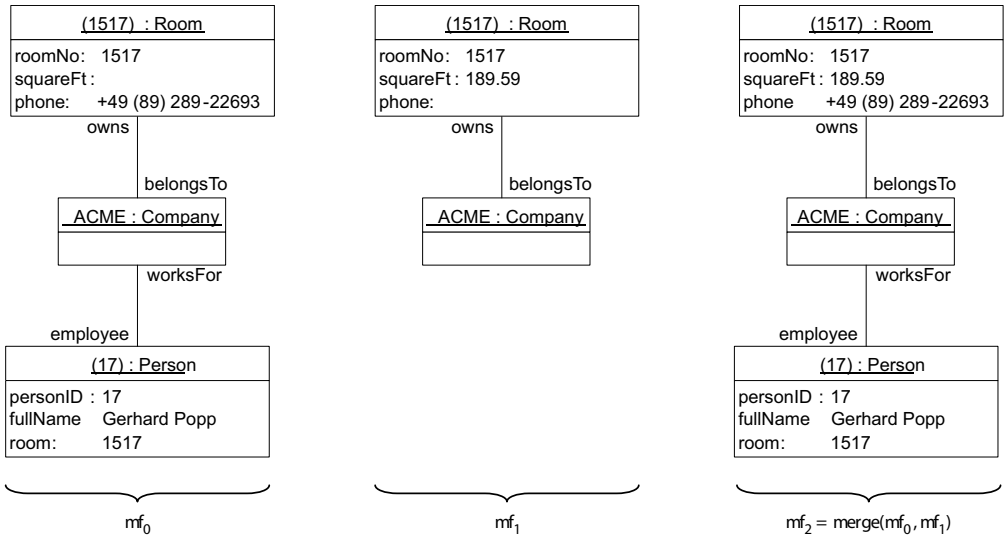


Abbildung 3.30: Zwei Modellfragmente und das Ergebnis der $\cup_{\mathfrak{m}}$-Operation

Beispiel: In Abbildung 3.30 sind drei Modellfragmente mf_0, mf_1, und mf_2 zu sehen, wobei gilt:

$$mf_2 = mf_0 \cup_{\mathfrak{m}} mf_1$$

Offensichtlich wurden durch das Zusammenführen der Modellfragmente die Objekte des Typs Room und Company jeweils verschmolzen, da sie über identische Primärschlüssel (roomNo = 1517) bzw.

Identifikatoren („ACME“ für Objekte des Typs Company) verfügen. Die Werte der Attribute squareFt und phone stammen jeweils nur aus einem Ursprungsobjekt, während das jeweilige Pendant an dieser Stelle den Wert $\diamondsuit$ aufweist. Das Objekt vom Typ Person in mf_0 wird, ebenso wie die Assoziation, die es mit dem restlichen Modellfragment verbindet, mit in das neu erzeugte Modellfragment übernommen. ❍

Zur besseren Lesbarkeit und um die Übersichtlichkeit zu verbessern, werden einige wesentliche Eigenschaften von Definition 3.5.12 im folgenden *Merge-Lemma* zusammengefasst:

Lemma 3.5.10 (Merge-Lemma)
Seine $mf_0, mf_1 \in \mathbb{MF}$ zwei Modellfragmente. Für $mf_2 = mf_0 \cup_\mathfrak{m} mf_1$ mit $mergeable(mf_0|_{OB}, mf_1|_{OB})$ gilt:

(*i*) $mf_2|_{OA}$ enthält alle Objektassoziationen, die in $mf_0|_{OA}$ oder $mf_1|_{OA}$ vorkommen. Falls Objektassoziationen in beiden Modellfragmenten vorkommen, so sind nur die mit der höheren Kardinalität *card* in $mf_2|_{OA}$ enthalten.

(*ii*) $mf_2|_{OA} \subseteq mf_0|_{OA} \cup mf_1|_{OA}$ (Beweis s. A.1.2, S. 283) ❑

In den beiden folgenden Sätzen wird festgehalten, dass die $\cup_\mathfrak{m}$-Operation kommutativ und assoziativ ist. Der Beweis basiert jeweils auf der Kommutativität und Assoziativität von $\cup$. Diese beiden Eigenschaften sind innerhalb des BOTL-Formalismus von großer Bedeutung, da durch sie sichergestellt ist, dass das Ergebnis einer Modelltransformation invariant gegenüber der Reihenfolge der Regelanwendungen und der Auswahl der Matches in den Quellmodellen ist.

Satz 3.5.1 (Kommutativität von $\cup_\mathfrak{m}$)
Die $\cup_\mathfrak{m}$-Operation ist kommutativ. (Beweis s. A.1.2, S. 284) ❑

Satz 3.5.2 (Assoziativität von $\cup_\mathfrak{m}$)
Die $\cup_\mathfrak{m}$-Operation ist assoziativ. (Beweis s. A.1.2, S. 284) ❑

Satz 3.5.3 hält die Eigenschaft, dass die Reihenfolge der Anwendung der $\cup_\mathfrak{m}$-Operation keinen Einfluss auf das Ergebnis hat, noch einmal explizit fest:

Satz 3.5.3 (Reihenfolge der Anwendung von $\cup_\mathfrak{m}$-Operationen)
Seien $mf_1, \ldots, mf_n \in \mathbb{MF}$ Modellfragmente. Dann gilt alle Permutationen *perm* einer Folge $i = 1, ..., n$:

$$\begin{aligned} &(((mf_1 \cup_\mathfrak{m} mf_2) \cup_\mathfrak{m} \cdots) \cup_\mathfrak{m} mf_{n-1}) \cup_\mathfrak{m} mf_n \\ =\ &(((mf_{perm(1)} \cup_\mathfrak{m} mf_{perm(2)}) \cup_\mathfrak{m} \cdots) \cup_\mathfrak{m} mf_{perm(n-1)}) \cup_\mathfrak{m} mf_{perm(n)} \end{aligned}$$ (Beweis s. A.1.2, S. 284) ❑

Der $\cup_\mathfrak{m}$-Operator wird nun herangezogen, um eine Teilmodell-Relation, wie sie in Definition 2.2.11 eingeführt wurde, für objektorientierte Modelle zu definieren.

Definition 3.5.13 (Teilmodellfragment $mf_0 \trianglelefteq mf_1$)
Ein Modellfragment $mf_0 \in \mathbb{MF}$ heißt *Teilmodellfragment* eines Modellfragments $mf_1 \in \mathbb{MF}$, falls die Relation $mf_0 \trianglelefteq mf_1$ gilt. Diese ist folgendermaßen definiert:

$$\trianglelefteq : \mathbb{MF} \times \mathbb{MF} \to \mathbb{B}$$
$$\text{mit } mf_0 \trianglelefteq mf_1 :\Leftrightarrow \exists mf_\mu \in \mathbb{MF} : mf_0 \cup_\mathfrak{m} mf_\mu = mf_1$$ ❍

Es bleibt anzumerken, dass gemäß dieser Definition ein Modellfragment auch dann Teilmodellfragment eines anderen Modellfragments ist, falls es strukturell identisch ist, aber für einige Attribute den Wert $\diamondsuit$ anstatt eines konstanten Wertes aufweist.

3.5.4 Regel- und Regelwerksanwendung

Die in den vorangegangenen Abschnitten vorgestellten Mechanismen zum Auffinden von Matches, zur Transformation von Modellfragmenten und zum Zusammenführen neu erzeugter Modellfragmente stellen die Grundbausteine für die Anwendung einer Regeln dar. Formal ist eine Regelanwendung durch die Abbildung $apply$ definiert, die im Weiteren vorgestellt wird. Darüber hinaus ist durch die Abbildung $transform$ die Anwendung eines ganzen Regelwerks definiert.

Definition 3.5.14 (Regelanwendung ($apply(m, r_i)$))
Eine *Regelanwendung* ist eine Abbildung

$$apply : \mathbb{MF}_{mm_0} \times \mathbb{R}_{mm_0,mm_1} \rightarrow \mathbb{MF}_{mm_1}$$
$$apply(m, r_i) \mapsto \bigcup_{mfm_j \in MFM(m, r_i|_{mv_0})}^{\mathfrak{m}} mft(mfm_j, r_i)$$

❍

Der nachfolgende Satz formuliert eine wichtige Eigenschaft von BOTL: Innerhalb einer Regelanwendung wird das Zielmodell sukzessive aufgebaut, indem Matches der linken Regelseite im Quellmodell gefunden werden, entsprechende Zielmodellfragmente erzeugt, und diese nacheinander in das Zielmodell germerged werden. Gemäß Satz 3.5.4 spielt die Reihenfolge in der die Matches gefunden werden dabei keine Rolle für das Ergebnis. Dementsprechend spielt auch die Reihenfolge der Merge-Operationen keine Rolle.

Satz 3.5.4 (Reihenfolge der der Auswahl von Modellfragment-Matches im Quellmodell)
Sei $r \in \mathbb{R}$ eine Regel und $m \in \mathbb{M}$ ein Modell. Dann ist das Ergebnis einer Regelanwendung $apply(m, r_i)$ (Definition 3.5.14) invariant bezüglich der Reihenfolge der Auswahl der Modellfragment-Matches $mfm \in MFM(m, r_i|_{mv_0})$ (Definition 3.5.3). (Beweis s. A.1.2, S. 285) ❑

Demzufolge liefert eine Regelanwendung immer ein deterministisches Ergebnis für ein gegebenes Quellmodell, unabhängig von der verwendeten Pattern-Matching-Strategie, die für die Suche nach Matches im Quellmodell verwendet wird.

Innerhalb eines Regelwerks können Regeln unterschiedliche Quellmodelle haben. Um einfach auf das Quellmodell einer Regel zugreifen zu können, wird nun die Funktion *srcModel* eingeführt:

Definition 3.5.15 ($srcModel(M, r_i)$)
Die Funktion *srcModel* liefert zu einer Regel aus einer Menge von Modellen das (einzige) Modell, welches das gleiche Metamodell wie die linke Modellvariable der Regel hat.

$$srcModel : \wp(\mathbb{M}) \times \mathbb{R} \rightarrow \mathbb{M}$$
$$srcModel(M, r_i) \mapsto \begin{cases} m \in M : r_i|_{mv_0}|_{mm} = m|_{mm} & \text{falls } \exists_1 m \in M : r_i|_{mv_0}|_{mm} = m|_{mm} \\ \emptyset & \text{sonst} \end{cases}$$

❍

Aufbauend auf der Definition der Regelanwendung wird nun die Anwendung eines Regelwerks definiert. Zur Anwendung eines Regelwerks werden alle enthaltenen Regeln angewendet und die jeweils entstandenen Modellfragmente zusammengeführt.

Definition 3.5.16 (Regelwerksanwendung (*transform*$(\{m_0,\ldots,m_n\},R)$))
Eine *Regelwerksanwendung* ist eine Abbildung:

$$transform : \wp(\mathbb{M}) \times \mathbb{RW} \rightarrow \mathbb{MF}$$

$$transform(M,R) \mapsto \bigcup_{r\in R}^{\mathfrak{m}} apply(srcModel(M,r),r)$$

$\diamondsuit$-Werte in mf werden hierbei durch jeweils passende Default-Werte ersetzt. ❍

Eine Regelwerksanwendung kann dementsprechend als Eingabe eine beliebige Menge von Modellen erhalten, deren Metamodelle jedoch paarweise unterschiedlich sein müssen. Modelle, deren Metamodell keinem Metamodell einer Quellmodellvariable entsprechen, werden vom Regelwerk ignoriert. Existieren Regeln, die Modelle eines in der Eingabe nicht vorhanden Typs verarbeiten, so wird aufgrund der Definition 3.5.15 von *srcModel* so verfahren als sei das entsprechende Quellmodell leer.

Satz 3.5.5 (Reihenfolge der Anwendung von Regeln eines Regelwerks)
Das Ergebnis einer Regelwerksanwendung auf eine Menge von Quellmodellen ist invariant bezüglich der Reihenfolge der Anwendung der Regeln des Regelwerks. (Beweis s. A.1.2, S. 285) ❑

Satz 3.5.5 formuliert eine wichtige Eigenschaft von BOTL: Die Reihenfolge, in der die einzelnen Regeln eines Regelwerks angewendet werden beeinflusst nicht das Ergebnis der Regelwerksanwendung. Dies bedeutet, dass keine Abhängigkeiten zwischen den Regeln untereinander existieren, wodurch Regelwerke ohne weiteres modular zusammengesetzt werden können.

Beispiel: Im Rahmen des vorgestellten Formalismus stellt eine Modelltransformation eine Regelwerksanwendung dar, wie sie in Definition 3.5.16 spezifiziert ist. Man erhält das Ergebnis der Transformation des Regelwerks $R = \{r_0, r_1\}$ durch Anwendung der Abbildung $transform(m_\alpha, R)$. Im Folgenden werden die einzelnen Schritte für die Anwendung der Regel r_0 detailliert aufgeführt:

$$\begin{aligned}
transform(m_\alpha,R) &= apply(\underbrace{m_\alpha}_{(i)},\underbrace{r_0}_{(ii)}) \cup_{\mathfrak{m}} apply(\underbrace{m_\alpha}_{(i)},\underbrace{r_1}_{(iii)}) \\
&= mft(\underbrace{mfm_{0,1}}_{(iv)},r_0) \cup_{\mathfrak{m}} mft(\underbrace{mfm_{1,1}}_{(v)},r_0) \cup_{\mathfrak{m}} mft(\underbrace{mfm_{1,0}}_{(vi)},r_1) \\
&= \underbrace{mf_{0,0}}_{(vii)} \cup_{\mathfrak{m}} \underbrace{mf_{0,1}}_{(viii)} \cup_{\mathfrak{m}} \underbrace{mf_1}_{(ix)} \\
&= \underbrace{mf_0}_{(x)} \cup_{\mathfrak{m}} \underbrace{mf_{1,0}}_{(ix)} \\
&= \underbrace{mf_\beta}_{(xi)}
\end{aligned}$$

Legende:

- (i) siehe Seite 62, Abbildung 3.10
- (ii) siehe Seite 75, Abbildung 3.18
- (iii) siehe Seite 77, Abbildung 3.19
- (iv) siehe Seite 85, Abbildung 3.26 (a)
- (v) siehe Seite 85, Abbildung 3.26 (b)
- (vi) siehe Seite 88, Abbildung 3.27 links

(*vii*) siehe Seite 88, Abbildung 3.27 rechts
(*viii*) siehe Seite 91, Abbildung 3.28 (a)
(*ix*) siehe Seite 91, Abbildung 3.28 (b)
(*x*) siehe Seite 100, Abbildung 3.31
(*xi*) siehe Seite 100, Abbildung 3.32

Abbildung 3.31 zeigt das durch Anwendung der Regel r_0 erzeugte Modellfragment mf_0.

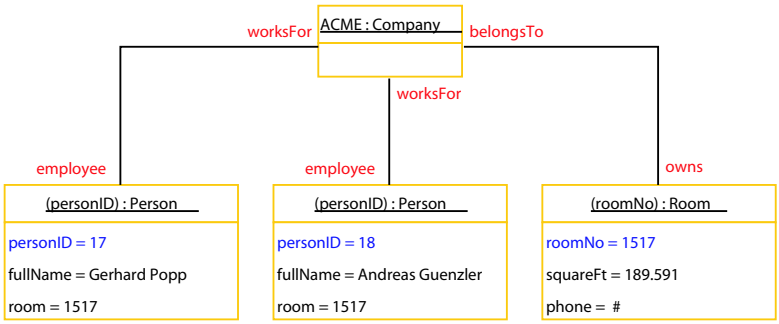


Abbildung 3.31: Das Ergebnis von $apply(m_\alpha, r_0) = mf_0$

Der Wert des Attributs phone wird erst durch die Anwendung der zweiten Regel erzeugt. Gemäß Satz 3.5.3 kann die Reihenfolge der *apply*-Anwendung der Regel auch vertauscht werden, ohne dass sich das Ergebnis hierdurch ändern würde.

Die zweite Regel r_1 wird in gleicher Weise angewendet. Durch den $\cup_m$-Operator wird das phone Attribut korrekt in alle existierenden Objekte des Typs Room geschrieben. Auch eine umgekehrte Reihenfolge der Regelanwendungen würde hierbei zu demselben Ergebnis führen. Das Ergebnis der Regelwerksanwendung ist in Abbildung 3.32 dargestellt.

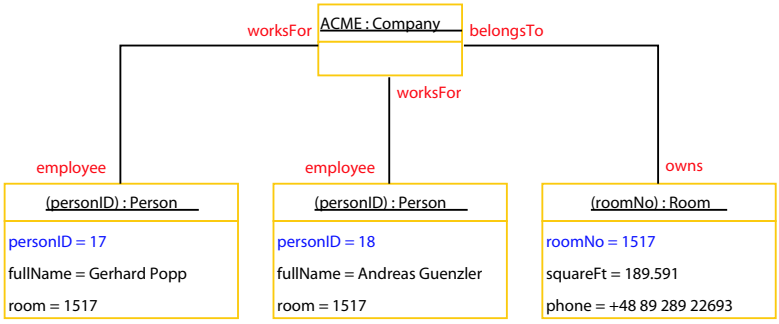


Abbildung 3.32: Das Ergebnis $mf_\beta := transform(\{m_\alpha\}, R$ der Anwendung des Beispielregelwerks R

❍

Ein interessanter Aspekt von BOTL ist die Möglichkeit, verschiedene Regelwerke mit demselben Zielmetamodell zu einem Regelwerk zu komponieren. Dies erlaubt es beispielsweise, Regelwerke für verschiedene Aspekte oder Quellen einer Transformation getrennt zu entwickeln und diese nach Bedarf zu kombinieren.

Der nachfolgende Satz besagt, dass für eine feste Menge von Quellmodellen und eine Menge von Regelwerken mit demselben Zielmetamodell das Zusammenführen der Ergebnisse der einzelnen Regelwerksanwendungen zu demselben Modellfragment führt, wie die Anwendung des Regelwerks, welches aus der Vereinigung der verschiedenen Regelwerke entsteht. Diese Aussage erlaubt es im

weiteren Verlauf der Arbeit, Aussagen über die Metamodellkonformität komponierter Modelle zu machen, wenn bekannt ist, aus welchen Regelwerksanwendungen sie entstanden sind.

Satz 3.5.6 (Parallelkomposition von Regelwerken)
Sei $\overline{R} = \{R_0, \ldots, R_n\} \in \mathscr{P}(\mathbb{RW})$ eine Menge von Regelwerken für die gilt:

$$\forall R_i, R_j \in \overline{R} : mm^1_{R_i} = mm^1_{R_j}$$

Weiter seien $M \in \mathbb{M}^0_{R_0 \cup \cdots \cup R_n}$ beliebige aber feste Mengen von Quellmodellen des Regelwerks $R_0 \cup \cdots \cup R_n$. Dann gilt:

$$transform(M, R_0) \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} transform(M, R_n) = transform(M, R_0 \cup \cdots \cup R_n)$$

(Beweis s. A.1.2, S. 285) ❑

Gemäß Satz 3.5.6 liefert also das Zusammenführen von Ergebnissen von Anwendungen verschiedener Regelwerke mit dem selben Zielmetamodell dasselbe Ergebnis wie die Anwendung des Regelwerks, welches aus der Vereinigung dieser Regelwerke hervorgeht. Vorraussetzung hierfür ist allerdings, dass alle Regelwerke dieselben Quellmodelle als Eingabe erhalten.

Im Nachstehenden Satz wird ein aus einer Transformation entstandenes Modell mit einer Menge von Modellen verglichen, welche aus Transformationen von Teilmodellen des Quellmodells durch das selbe Regelwerk entstanden sind. Dann gilt, dass auch die Komposition der aus den Teilmodellen entstandenen Zielmodelle immer ein Teilmodell der Transformation des gesamten Modells ist. Diese Eigenschaft kann beispielsweise genutzt werden, um nachzuweisen, dass eine Integrationstransformation eine konsistente Verfeinerung bezüglich einer Transformation darstellt (siehe auch Abschnitt 5.3.2, S. 215).

Satz 3.5.7 (Transformation von Teilmodellen)
Es sei $R \in \mathbb{R}$ ein Regelwerk mit einem einzigen Quellmetamodell mm und $m_0, \ldots, m_n \in \mathbb{M}_{mm}$ eine Menge möglicher Quellmodelle für die gilt $m_0 \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} m_n \neq \bot$. Weiter sei

$$mf := transform(m_0, R) \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} transform(m_n, R)$$

Dann gilt:

$$mf \neq \bot \Rightarrow mf \trianglelefteq transform(m_0 \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} m_n, R)$$

(Beweis s. A.1.2, S. 285) ❑

Die beiden Relationen *createsObj* und *createsAsso* geben an, ob ein Objekt bzw. eine Objektassoziation für ein gegebenes Quellmodell aus einer gegeben Objektvariable bzw. einer gegebenen Objektvariablenassoziation erzeugt wurde oder nicht. Diese Eigenschaften werden im weiteren Verlauf für den Nachweis von Eigenschaften von Regelwerken benötigt werden.

Definition 3.5.17 (*createsObj*(o, ov, r_i, m), *createsAsso*$(asso, ova, r_i, m)$)
Sei $R \in \mathbb{RW}$ ein Regelwerk, $M \in \mathbb{M}^0_R$ eine Menge von Quellmodellen des Regelwerks, $r_i \in R$ und $m := srcModel(M, r_i)$.

Ein Objekt $o \in \mathbb{O}$ wurde von einer Objektvariable $ov \in r_i|_{OVB}$ erzeugt, falls die Relation *createsObj* gilt:

$$
\begin{aligned}
&\mathit{createsObj} : \mathbb{O} \times \mathbb{OV} \times \mathbb{R} \times \mathbb{M} \to \mathbb{B} \\
&\mathit{createsObj}(o, ov, r_i, m) :\Leftrightarrow \exists mfm \in MFM(m, r_i|_{mv_0}) : \\
&\qquad\qquad \big(o \in mft(mfm, r_i)|_{OB} \wedge \\
&\qquad\qquad \exists mfm^1_\mu \text{ gemäß Def. 3.5.4 } : mfm^1_\mu|_{match_o}(ov) = o\big)
\end{aligned}
$$

Eine Assoziation $asso \in \mathbb{OA}$ wurde von einer Objektvariablenassoziation $ova \in r_i|_{OVA}$ erzeugt, falls die Relation *createsAsso* gilt:

$$
\begin{aligned}
&\mathit{createsObj} : \mathbb{OA} \times \mathbb{OVA} \times \mathbb{R} \times \mathbb{M} \to \mathbb{B} \\
&\mathit{createsAsso}(asso, ova, r_i, m) :\Leftrightarrow \exists mfm \in MFM(m, r_i|_{mv_0}) : \\
&\qquad\qquad \big(asso \in mft(mfm, r_i)|_{OA} \wedge \\
&\qquad\qquad \exists mfm^1_\mu \text{ gemäß Def. 3.5.4 } : mfm^1_\mu|_{match_a}(ova) = asso\big) \quad \circ
\end{aligned}
$$

Bis zu diesem Punkt der Arbeit wurde ein mathematisches Modell für Klassen, Objekte und Regelwerke erarbeitet und spezifiziert, wie Regelwerke angewandt werden. Dennoch kann die Anwendung eines Regelwerks nach wie vor $\perp$ als Ergebnis liefern, was bedeutet, dass der Transformationsprozess nicht erfolgreich durchgeführt werden konnte. In den folgenden Abschnitten werden Techniken vorgestellt, mit denen sich zeigen lässt, dass ein Regelwerk immer erfolgreich angewendet werden kann und ggf. immer ein zum Zielmetamodell konformes Modell erzeugt.

3.6 Rewrite-Transformationen

Der bisher vorgestellte Ansatz erlaubt lediglich die Transformation einer Menge von Quellmodellen in ein gemeinsames Zielmodell. Hierbei wird das Zielmodell jeweils neu erzeugt. In vielen Bereichen wird jedoch kein neues Modell erzeugt, sondern ein bestehendes Modell durch die Anwendung von Operationen schrittweise verändert. Ein Beispiel hierfür sind die in Abschnitt 2.2.2 vorgestellten Prozessmuster, deren Anwendung jeweils ein Erweiterung des Modells der Artefakte um neu geschaffene Artefakte und Beziehungen zwischen Artefakten bewirkt. Um diesen Anforderungen genügen zu können, wird der BOTL-Ansatz im Folgenden um die Möglichkeit erweitert, in bestehende Quellmodelle schreiben zu können.

Der vorliegende Ansatz beschränkt sich hierbei auf eine beherrschbare Klasse von Modellen und unterstützt lediglich schreibende Operationen auf diesen Modellen. Die für diese Klasse von Modellen geltenden Einschränkungen sind zum einen, dass es keine zwei Objektassoziationen gleichen Typs zwischen zwei identischen Objekten in der gleichen Richtung geben darf. Diese Forderung wird so auch für Instanzen von MOF-Metamodellen erhoben, d.h. für MOF-basierte Modelle ist durch diese zusätzliche Regelung keine echte Einschränkung gegeben. Weiterhin dürfen die hier betrachteten Modelle keine reflexiven Objektassoziationen oder „freischwebende“ Objekte, d.h. Objekte die über keinerlei Assoziationen mit dem anderen Objekten verbunden sind, enthalten.

Die Klasse dieser Modelle und Modellfragmente wird als die Klasse der C1-Modelle bezeichnet. Die folgende Definition hält führt diesen Begriff formal ein:

Definition 3.6.1 (C1-Modellfragmente)
Ein *C1-Modellfragment* ist ein Modellfragment $mf \in \mathbb{MF}$ für das im Folgenden definierte Prädikat $C1$ gilt.

$$C1 : \mathbb{MF} \rightarrow \mathbb{B}$$
$$C1(mf) :\Leftrightarrow \forall oa \in mf|_{OA} : (oa|_{card} = 1 \ \wedge \ \left|oa|_{OAE}\right| = 2) \quad \wedge$$
$$\forall o \in mf|_{OB} : (\exists oa \in mf|_{OA} : o \in oa|_{OAE}|_o)$$

$\mathbb{M}'_{mm}$ bezeichnet die Menge aller zu einem Metamodell *mm* konformen C1-Modelle. ❍

Für die Praxis sind die Restriktionen von C1-Modellen jedoch unbedeutend, da redundante oder reflexive Assoziationen in realen Modelle kaum aufzufinden sind. Freischwebende Objekte sind über keinen Navigationspfad erreichbar und somit ebenfalls von keiner praktischen Relevanz.

Dementsprechend wird eine C1-Regel als eine Regel definiert, deren Zielmodellvariable die gleichen Eigenschaften wie C1-Modellfragmente aufweist.

Definition 3.6.2 (C1-Regeln)
Eine C1-Regel $r_i \in \mathbb{R}$ ist eine Regel, für die das im Folgenden definierte Prädikat $C1$ gilt:

$$C1 : \mathbb{R} \rightarrow \mathbb{B}$$
$$C1(r_i) :\Leftrightarrow \forall ova \in r_i|_{mv_1}|_{OVA} : (ova|_{cardv} = 1 \ \wedge \ \left|ova|_{OVAE}\right| = 2) \quad \wedge$$
$$\forall ov \in r_i|_{mv_1}|_{OVB} : (\exists ova \in r_i|_{mv_1}|_{OVA} : ov \in ova|_{OVAE}|_o v)$$ ❍

Im Folgenden wird festgehalten, dass C1-Regeln ausschließlich C1-Modellfragmente erzeugen können.

Lemma 3.6.1 (C1-Regeln erzeugen C1-Modellfragmente)
C1-Regeln erzeugen nur C1-Modellfragmente:

$$\forall r \in \mathbb{R}, m \in \mathbb{M}_{r|_{mv_0}|_{mm}} \text{ mit } C1(r) : C1(apply(m,r))$$ (Beweis s. A.1.3, S. 286) ❑

BOTL-Rewrite-Transformationen erweitern ein bestehendes Modell schrittweise um neue Modellelemente. Um die bestehenden Verifikationstechniken für den Nachweis der Anwendbarkeit und Metamodellkonformität von BOTL-Transformationen auch für Rewrite-Transformationen nutzbar machen zu können, werden Rewrite-Transformationen als ein Sonderfall gewöhnlicher BOTL-Transformationen behandelt. Während bei BOTL-Transformationen ein neues Zielmodell aus einer Menge von Quellmodellen aufgebaut wird, werden durch Rewrite-Transformationen bestehende Modelle erweitert. Demnach entspricht eine Rewrite-Transformation einer BOTL-Transformation, die

1. *ein* ausgewähltes Modell vollständig kopiert und
2. das so erzeugte Modell durch eine Reihe weiterer BOTL-Transformationen erweitert.

Um Modelle vollständig kopieren zu können werden im Folgenden C1-Identitäts-Regelwerke eingeführt. Ein solches Regelwerk legt eine Kopie eines C1-Modell an, wobei Quell- und Zielmetamodell des Regelwerks isomorph sind.

Definition 3.6.3 (C1-Identitäts-Regelwerk $R^{id}_{mm_0}$)
Zu einem gegebenen Metamodell $mm_0 \in \mathbb{MM}$ ist $R^{id}_{mm_0} \in \mathbb{RW}$ das *C1-Identitäts-Regelwerk* für das gilt:

(i) das Zielmetamodell mm_1 ist isomorph zu mm_0:
$\exists mm_1 \in \mathbb{MM} : mm_1 \cong mm_0 \wedge$
$mm_1 \neq mm_0$

(ii) Es existiert eine unendliche Menge unterschiedlicher Identifikatoren

$$\{id_0^0, \ldots, id_0^3, id_1^0, \ldots, id_1^3, \ldots\} \subset \mathbb{ID}$$

(iii) Für jede Klassenassoziation in mm_0 existiert genau eine *Assoziationsregel*:

$\forall asso_i \in mm_0|_{CA}$ mit $asso_i = \{ae_0, ae_1\}$:
$\exists_1 r_i := (mv_0, mv_1) \in R^{id}_{mm_0}$:
$mv_0 := \big(mm_0, \{ov_0, ov_0'\}, \{(asso_i, 1, \{(ae_0, ov_0), (ae_1, ov_0')\})\}\big)$
$mv_1 := \big(mm_1, \{ov_1, ov_1'\}, \{(asso_i, 1, \{(ae_0, ov_1), (ae_1, ov_1')\})\}\big)$
wobei gilt:
$ov_0 := (id_i^0, oiv, otv, VV)$
$ov_1 := (id_i^1, oiv, otv, VV)$
$oiv := \begin{cases} objID \in \mathbb{VAR} \setminus \{oiv'\} & \text{falls } ae_0|_c|_{Keys} = \emptyset \\ \varepsilon & \text{sonst} \end{cases}$
$otv := ae_0|_c$
$\forall (n_k, t_k) \in ae_0|_c|_A$:
$\exists_1 (a_k, t_k) \in VV$:
$a_k = n_k$
$t_k \in \mathbb{VAR} \setminus (\{oiv, oiv'\} \cup ov_0'|_{VV}|_t)$ mit
$\nexists (a_k, t_k), (a_j, t_j) \in VV : t_k = t_j \wedge (a_k, t_k) \neq (a_j, t_j)$
$ov_0' := (id_i^2, oiv', otv', VV')$
$ov_1' := (id_i^3, oiv', otv', VV')$
$oiv' := \begin{cases} objID' \in \mathbb{VAR} \setminus \{oiv\} & \text{falls } ae_0|_c|_{Keys} = \emptyset \\ \varepsilon & \text{sonst} \end{cases}$
$otv' := ae_1|_c$
$\forall (n_k', t_k') \in ae_1|_c|_A$:
$\exists_1 (a_k', t_k') \in VV'$:
$a_k' = n_k'$
$t_k' \in \mathbb{VAR} \setminus (\{oiv, oiv'\} \cup ov_0|_{VV}|_t)$ mit
$\nexists (a_k', t_k'), (a_j', t_j') \in VV' : t_k' = t_j' \wedge (a_k', t_k') \neq (a_j', t_j')$ ❍

Beispiel: Abbildung 3.33 zeigt das Metamodell mm_0. Dann besteht das zugehörige C1-Identitäts-Regelwerk $R^{id}_{mm_0}$ für dieses Metamodell gemäß Definition 3.6.3 aus den in in Abbildung 3.34 dargestellten Regeln.

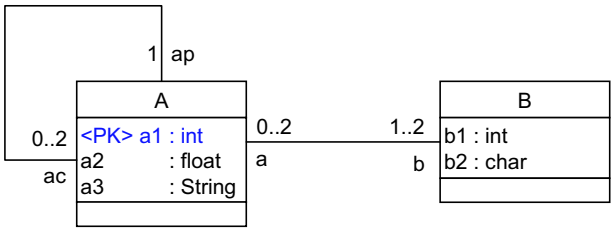


Abbildung 3.33: Das Metamodell mm_0

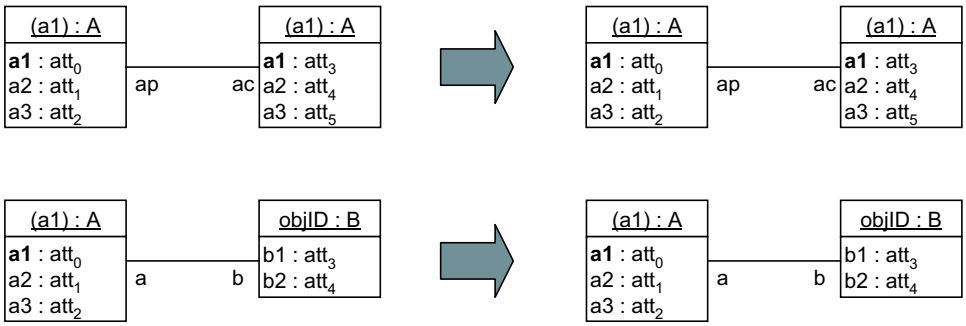


Abbildung 3.34: Das Regelwerk $R^{id}_{mm_0}$

Für jede Klassenassoziation im Metamodell werden Regeln generiert, die alle Assoziationen mit der Kardinalität 1 erfassen und die jeweilige Objektassoziation zusammen mit den verbunden Objekten kopieren. Hierbei werden sämtliche Attribute und die Identifikatoren der beiden Objekte kopiert. ❍

Die in Rewrite-Regelwerken enthaltenen Regeln sind prinzipiell identisch mit denen in „klassischen“ BOTL-Regelwerken. Der wesentliche Unterschied besteht jedoch darin, dass Rewrite-Regelwerke auch Regeln enthalten dürfen, deren Quell- und Zielmetamodell identisch ist. Dementsprechend operieren diese Regeln auf einem Modell, welches zugleich als Quell- und Zielmodell dient. In der nachfolgenden Definition von Rewrite-Regelwerken wird zudem festgelegt, dass eine Menge von Regeln in Form einer „Transaktion“ ausgeführt werden, d.h. alle Änderungen des Zielmodells werden für Rewrite-Regeln erst der Ausführung aller Regeln dieser Menge sichtbar.

Definition 3.6.4 (Rewrite-Modelltransformationsregelwerk *RRW*)
Ein *Rewrite-Modelltransformationsregelwerk* (oder kurz: Rewrite-Regelwerk) *RW* ist eine endliche Menge von Modelltransformationsregeln $\{r_0, ..., r_n\} \in \mathscr{P}(\mathbb{R})$ für die gilt:

$$\exists_1 mm_1 \in \mathbb{MM} : \quad \forall r \in RW :$$
$$\big(r|_{mv_1}|_{mm} = mm_1 \ \wedge \tag{3.25}$$
$$C1(r) \qquad\qquad \big) \tag{3.26}$$

Regeln $r \in RW$ für die gilt

$$r|_{mv_0}|_{mm} = r|_{mv_1}|_{mm}$$

werden als *Rewrite-Regeln* bezeichnet. Die Menge aller denkbaren Rewrite-Modelltransformationsregelwerke wird mit $\mathbb{RRW}$ bezeichnet. Weiterhin existiert eine Abbildung tx mit

$$tx : \mathbb{RRW} \times \mathbb{R} \to \mathbb{N}_0^+ \text{ , mit}$$
$$\forall r \notin RW : tx(RW, r) = \bot$$

Für ein Rewrite-Regelwerk RW wird die folgende Schreibweise eingeführt, um Mengen von Regeln mit gleichem tx-Wert zu referenzieren:

$$RW^i := \{r \in RW : tx(RW, r) = i\} \qquad \text{❍}$$

(3.25) legt fest, dass alle Regeln eines Rewrite-Modelltransformationsregelwerks Abbildungen mit demselben Zielmetamodell mm_1 definieren. Im Gegensatz zu gewöhnlichen Regelwerken kann bei Rewrite-Regelwerken auch das Zielmodell als Quellmodell verwendet werden. (3.26) besagt, dass alle Regeln eines Rewrite-Regelwerks C1-Regeln gemäß Definition 3.6.2 sein müssen.

Die Abbildung tx ordnet Regeln eines Regelwerks eine Zahl aus $\mathbb{N}_0^+$ zu. Regeln mit demselben tx-Wert werden in einer Art Transaktion durchgeführt, d.h. erst nach Anwendung *aller* Regeln mit einem identischen tx-Wert können wieder Regeln mit einem anderen tx-Wert ausgeführt werden.

Im Rahmen einer Rewrite-Regelwerksanwendung werden eine Menge von Quellmodellen in ein Zielmodell transformiert, wobei das Zielmodell in der Menge der Quellmodelle enthalten sein darf. Regeln mit einer identischen tx-Zahl werden erst gemeinsam vollständig angewendet und ihr Ergebnis dann mit dem Zielfragment zusammengeführt.

Da durch Rewrite-Regeln im Verlauf der Transformation ein Quellmodell verändert wird, hängt das Ergebnis einer solchen Transformation von der Reihenfolge der Anwendung der einzelnen Regeln ab. Durch eine Sequenz aus natürlichen Zahlen wird für eine Regelwerksanwendung die Reihenfolge der Anwendung der Regeln mit identischen tx-Wert festgelegt. Die nachfolgende Definition legt die Auswirkung der Anwendung von Rewrite-Regelwerken formal fest.

Definition 3.6.5 (Rewrite-Regelwerksanwendung (*rwTransform*$(M, RW, (t_0, \ldots, t_n))$))
Sei $RW \in \mathbb{RRW}$ ein Rewrite-Regelwerk mit dem Zielmetamodell $mm_n := mm^1_{RW}$, $M \in \mathscr{P}(\mathbb{M})$ eine Menge von Modellen mit $m_n \in M \wedge m_n|_{mm} = mm_n$ und $(t_0, t_1, \ldots t_n)$ eine Sequenz natürlicher Zahlen. Dann ist eine *Rewrite-Regelwerksanwendung* eine Abbildung mit:

$$rwTransform : \wp(\mathbb{M}) \times \mathbb{RRW} \times (\mathbb{N}_0^+ \times \ldots \times \mathbb{N}_0^+) \to \mathbb{MF}$$
$$rwTransform(M, RW, (t_0, t_1, \ldots t_n)) :=$$
$$\begin{cases} \bot & \text{falls } \neg C1(m_n) \\ rwTransform\big(transform(M, R^{id}_{mm} \cup RW^{t_0}) \cup M \backslash m_n, & \\ \qquad\qquad RW, (t_1, \ldots, t_n)\big) & \text{falls } C1(m_n) \wedge |(t_1, \ldots, t_n)| > 0 \\ m \in M \text{ mit } m_{mm} = mm_n & \text{sonst} \end{cases}$$

Die Folge natürlicher Zahlen $(t_0, \ldots t_n)$ wird hierbei *Ausführungsfolge* der Transformation genannt. ❍

Eine Rewrite-Regelwerksanwendung liefert demnach nur dann ein Ergebnis ungleich $\bot$, falls das Zielmodell ein C1-Modell ist. Im Verlauf einer Rewrite-Regelwerkanwendung werden der Reihe nach die Regeln mit dem aktuellen tx-Wert der Ausführungsfolge ausgeführt und das Ergebnis jeweils in

das Zielmodellfragment eingefügt. Für die Ausführung einer Menge von Regeln mit identischem *tx*-Wert wird die *transform*-Abbildung für herkömmliche BOTL-Regelwerke verwendet. Diese Abbildung ist auch für Rewrite-Regeln definiert, die Einschränkung, dass Quell- und Zielmetamodell bei herkömmlichen BOTL-Transformationen unterschiedlich sein müssen gilt lediglich für die BOTL-Regelwerke (siehe Definition 3.3.10.

Der durch die *tx*-Funktion geschaffene Transaktionsmechanismus kann verwendet werden, um sicherzustellen, dass nach der Anwendung einer Menge von *tx*-Regeln das Zielmodell wieder ein konsistentes C1-Modell ist. Wäre dies nicht gewährleistet, so würden nachfolgende Rewrite-Regeln ein inkonsistentes Modell als Quellmodell vorfinden, was ihre Anwendung nicht möglich macht.

4 Eigenschaften von BOTL-Regelwerken

Im vorangegangen Kapitel wurde ein mathematisch fundiertes Modell für objektorientierte Metamodelle und Modelle sowie der darauf basierende Transformationsmechanismus BOTL vorgestellt. Wie bereits in Kapitel 2.2 angeführt wurde, ist der Einsatz eines formal fundierten Mechanismus zur Modelltransformation nur dann sinnvoll, falls es möglich ist, für eine Transformationsspezifikationen eine Reihe von Eigenschaften von sicherzustellen, bzw. diese formal verifizieren zu können.

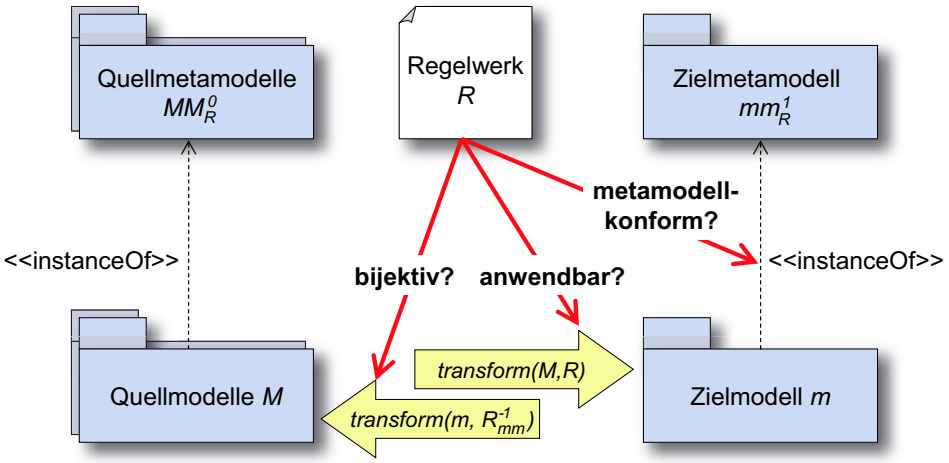


Abbildung 4.1: Eigenschaften von BOTL-Regelwerken

In den nachstehenden Unterabschnitten werden zunächst die drei in Abbildung 4.1 skizzierten Eigenschaften für BOTL-Regelwerken eingeführt. Im Einzelnen sind dies:

Anwendbarkeit von Regelwerken Ein Regelwerk ist *anwendbar*, falls sichergestellt werden kann, dass seine Anwendung immer gültige Modellfragmente erzeugt. D.h. seine Anwendung darf für beliebige, gültige Quellmodelle niemals den Fehlerwert $\perp$ als Ergebnis liefern.

Metamodellkonformität Bei der Anwendung eines *metamodellkonformen* Regelwerks ist sichergestellt, dass es für beliebige, gültige Quellmodelle immer ein Zielmodell erzeugt, welches konform zu dem Zielmetamodell des Regelwerks ist. Im Gegensatz zu gültigen Modellfragmenten müssen in gültigen Modellen auch die in ihrem jeweiligen Metamodell definierten Assoziationsober- und -untergrenzen eingehalten werden.

Bijektivität Bei *bijektiven* Regelwerken lassen sich für ein Quellmetamodell alle Modellvariablen in Regeln die auf Instanzen dieses Metamodells zugreifen vertauschen. Hierbei ist sichergestellt, dass die Anwendung dieses Umkehrregelwerks aus einem erzeugten Zielmodell immer wieder das Quellmodell oder aber ein zum Quellmodell isomorphes Modell erzeugt.

Für jede dieser Eigenschaften werden Verifikationsmechanismen vorgestellt, die es erlauben, die jeweilige Eigenschaft eines Regelwerks formal zu verifizieren. Die vorgestellten Verifikationstechniken

sind in der Regel hinreichend um die gewünschten Eigenschaften nachzuweisen, jedoch nicht notwendig. Durch eine geeignete Werkzeugunterstützung ist es möglich anhand der hier vorgestellten Verfifikationstechniken die gewünschten Eigenschaften von Regelwerken automatisiert nachzuweisen. In Kapitel 6 ist eine entsprechende Werkzeugunterstützung vorgestellt.

Darüber hinaus wird in Abschnitt 4.4 eine Technik vorgestellt, die es ermöglicht, auch für Rewrite-Regelwerke nachzuweisen, dass diese ggf. metamodellkonform sind. Abschnitt 4.5 diskutiert anhand eines Beispiels knapp, wie Aussagen über die Semantik von BOTL-Regelwerken machen lassen und innerhalb von Abschnitt 4.6 werden die Eigenschaften des BOTL-Ansatzes anhand des Klassifikationsschemas aus Abschnitt 2.2.1 zusammengefasst.

4.1 Anwendbarkeit von Regelwerken

Die *Anwendbarkeit* von Regelwerken ist eine Eigenschaft, die aussagt, dass die Anwendung eines gegebenen Regelwerks für beliebige aber konforme Quellmodelle zu einem Ergebnis ungleich $\perp$ führt. Zunächst wird der Begriff Anwendbarkeit formal gefasst:

Definition 4.1.1 (Anwendbarkeit von Regelwerken)
Ein Regelwerk $R \in \mathbb{RW}$ heißt genau dann *anwendbar*, wenn seine Anwendung für beliebige gültige Mengen von Quellmodellen immer gültige Modellfragmente (d.h. $\neq \perp$) erzeugt:

$$R \in \mathbb{RW} \text{ ist anwendbar } :\Leftrightarrow \forall M \in \mathbb{M}_R^0 : \mathit{transform}(M,R) \neq \perp \qquad \text{❍}$$

Um die Anwendbarkeit eines Regelwerks leichter verifizieren zu können, wird der Begriff Anwendbarkeit zunächst für einzelne Regeln definiert. Eine Regel ist, ebenso wie ein Regelwerk, anwendbar, falls ihre Anwendung niemals den Wert $\perp$ als Ergebnis liefert.

Definition 4.1.2 (Anwendbarkeit einer Regel)
Eine Regel $r \in \mathbb{R}$ ist genau dann *anwendbar*, falls ihre Anwendung ein gültiges (d.h. $\neq \perp$) Modellfragment erzeugt:

$$r \in \mathbb{R} \in \text{ ist anwendbar } :\Leftrightarrow \forall m \in \mathbb{M}_{r|_{mv_0}|_{mm}} : \mathit{apply}(m,r) \neq \perp \qquad \text{❍}$$

Wie im Verlauf dieses Abschnitts gezeigt wird (siehe Satz 4.1.6, S. 130 und Satz 4.1.7, S. 130), existieren im Wesentlichen existieren drei Fälle, die dazu führen können, dass die Anwendung eines Regelwerks zu dem Ergebnis $\perp$ führt:

1. Eine einzelne Modellfragmenttransformation erzeugt ein ungültiges Modellfragment. Dieser Fall wird in Abschnitt 4.1.1 behandelt.

2. Im Verlauf der Anwendung einer Regel erzeugt eine Objektvariable mehrfach dasselbe Objekt mit widersprüchlichen Attributwerten. Abschnitt 4.1.2 stellt Techniken zur Erkennung solcher potentieller Fehlerquellen vor.

3. Im Verlauf der Anwendung eines Regelwerks erzeugen verschiedene Objektvariablen (aus potentiell unterschiedlichen Regeln) dasselbe Objekt mit widersprüchlichen Attributwerten. Dieser Fall wird innerhalb von Abschnitt 4.1.3 diskutiert.

Die folgenden Abschnitte führen eine Reihe von Eigenschaften von Regeln und Verifikationstechniken für sie ein. Diese werden benötigt, um die oben genannten Konfliktfälle auszuschließen und die Anwendbarkeit von Regeln und Regelwerken verifizieren zu können. Die vorgestellten Techniken erlauben es, hinreichende Aussagen darüber zu machen, ob ein Regelwerk anwendbar ist. Ziel ist es zum einen, eine möglichst große Klasse von anwendbaren Regelwerken als solche identifizieren zu können, und zum anderen, Techniken zu spezifizieren, die auch einen automatisierten Nachweis der Anwendbarkeit von Regelwerken erlauben.

4.1.1 Erzeugen gültiger Modellfragmente

Um sicherzustellen, dass eine Regel anwendbar ist, gilt es zunächst sicherzustellen, dass keine Transformation der Regel für ein konformes Quellmodell jemals den Wert $\perp$ als Ergebnis liefert. Die nachfolgende Definition führt das Prädikat *createsValidFragments* ein, welches genau dann gilt, falls bei einem beliebigen, gültigen Quellmodell jede Anwendung einer Regel r ein Ergebnis ungleich $\perp$ erzeugt wird.

Definition 4.1.3 (*createsValidFragments*(r))
Sei $r \in \mathbb{R}$ eine Regel. Das Prädikat *createsValidFragments*(r) gilt genau dann, falls r nur gültige Modellfragmente erzeugt:

$$\begin{aligned}
&\mathit{createsValidFragments} : \mathbb{R} \to \mathbb{B} \text{ mit} \\
&\mathit{createsValidFragments}(r) :\Leftrightarrow \\
&\quad \forall m \in \mathbb{M}|_r|_{mv_0}|_{mm}, mfm \in \mathit{MFM}(m, r|_{mv_0}) : \mathit{mft}(mfm, r) \neq \perp
\end{aligned}$$

❍

Beispiel: Für eine Regel r gilt das Prädikat *createsValidFragments*(r) immer dann, wenn sichergestellt ist, dass jede Modellfragmenttransformation ein gültiges Modellfragment ungleich $\perp$ erzeugt. Abbildung 4.2 zeigt eine Regel, die offensichtlich *keine* gültigen Modellfragmente erzeugt.

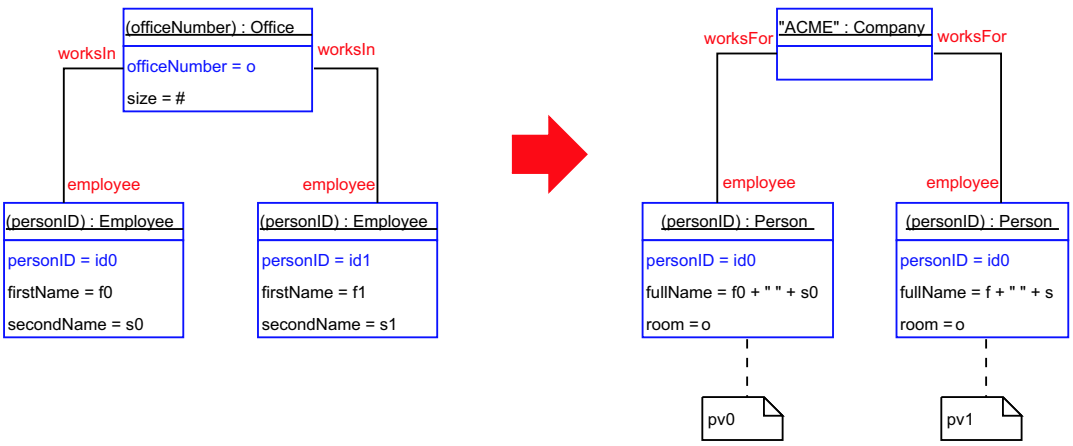


Abbildung 4.2: Beispiel für eine Regel die *keine* gültigen Modellfragmente erzeugt.

Die dargestellte Regel durchsucht ein Quellmodell nach Objekten des Typs Office, die mit jeweils zwei Employee-Objekten verbunden sind und erzeugt hieraus zwei Objekte des Typs Person an demselben Company-Objekt. Das Ergebnis jeder Modellfragmenttransformation dieser Regel ist $\perp$. Hierfür gibt es zwei Gründe:

1. Die beiden Zielmodellvariablen $pv0$ und $pv1$ haben in ihrem einzigen Primärschlüsselattribut personID beide dieselbe Variable $id0$ stehen. Demzufolge enthält jedes erzeugte Fragment zwei Objekte des Typs Person mit identischen Werten in ihren Primärschlüsselattributen. Die beide Objekte haben also dieselbe Identität $pK(\{(\mathsf{personID}, id0)\})$.

   Da zwei solche Objekte innerhalb einer Objektbelegung nicht existieren dürfen (siehe Definition 3.1.12 (3.12)), stellt das Ergebnis einer Modellfragmenttransformation *kein* gültiges Modellfragment dar. Dementsprechend wird stattdessen $\perp$ als Ergebnis gelierfert.

2. Unabhängig von den identischen Werten in Primärschlüsselattributen existiert jedoch noch ein Grund dafür, dass die vorgestellte Regel keine gültigen Modellfragmente erzeugt: Der Wert des Attributs fullName errechnet sich für die von der Objektvariablen $pv1$ erzeugten Objekte durch den Ausdruck $f + \text{„ “} + s$. Da die beiden Variablen f und s jedoch nicht weiter gebunden sind, ist für sie jeder String-Wert eine gültige Lösung. Demzufolge verfügt das Gleichungssystem *GL* für diese Regel über mehr als eine Lösung und die Modellfragmenttransformation mft liefert gemäß Definition 3.5.5 das Ergebnis $\perp$.

In diesem Fall führt also *jede* Modellfragmenttransformation dieser Regel zu dem Ergebnis $\perp$. Allgemeinen gilt das Prädikat *createsValidFragments* jedoch nur dann für eine Regel *r*, falls mit Sicherheit *keine einzige* denkbare Modellfragmenttransformationen der Regel *r* das Ergebnis $\perp$ liefert. ❍

Das nachfolgende Lemma 4.1.1 hält fest, dass eine Modellfragmenttransformation immer dann das Ergebnis $\perp$ liefert, wenn entweder mehr als eine Lösung für die Modellfragmentrelation mfr existiert, oder das erzeugte Fragment kein konsistentes Fragment bezüglich seines Metamodells ist.

Lemma 4.1.1 ($\perp$ als Ergebnis einer Modellfragmenttransformation)
Seien $mm_0, mm_1 \in \mathbb{MM}$ das Quell- bzw. Zielmetamodell einer Regel $r \in \mathbb{R}_{mm_0,mm_1}$ und $mfm_0 \in MFM(mf_0, r|_{mv_0})$ mit $mf_0 \in \mathbb{MF}_{mm_0}$ ein Modellfragment-Match. Dann gilt:

$$mft(mfm_0, r) = \perp \quad \Leftrightarrow \quad \left|\{mf_1 \in \mathbb{MF} : mfr(mfm_0, r, mf_1)\}\right| > 1 \vee \tag{4.1}$$

$$\exists\, mf_1 \in \mathbb{MF} : mfr(mfm_0, r, mf_1)) \wedge mf_1 \notin \mathbb{MF}|_{mm_1} \tag{4.2}$$

(Beweis s. A.2.1, S. 287) ❑

Für jede Modellfragmenttransformation wird durch das Gleichungssystem *GL* aus Definition 3.5.4 der Zusammenhang zwischen den im Quellmodell gefundenen Attribut- und Identifikatorwerten charakterisiert. Zieht man nun zusätzlich in Betracht, dass die Quellmodelle einer Modelltransformation Modelle gemäß Definition 3.1.14 sein müssen, so impliziert dies, dass ihre Objekte jeweils eine Objektbelegung (siehe Definition 3.1.12) bilden müssen. Dementsprechend müssen alle Objekte gleichen Typs, die im Verlauf einer Modellfragmenttransformation gematcht werden, unterschiedliche Identifikatoren bzw. Primärschlüssel aufweisen.

Das im Folgenden definierte Ungleichungssystem $UGL_{validSrc}$ charakterisiert formal diesen Constraint. Die hiermit verbundenen, stärkeren Aussagen über die Lösung des Gleichungssystems *GL* werden benötigt, um nachweisen zu können, dass eine Regel gültige Modellfragmente erzeugt.

Definition 4.1.4 (Ungleichungssystem für gültige Quellfragmente $UGL_{validSrc}$)
Sei $r \in \mathbb{R}$ eine Regel und *GL* das Gleichungssystem aus Definition 3.5.4. Dann ist das Gleichungssys-

tem $UGL_{validSrc}$ definiert als:

$$GL \qquad \wedge \qquad (4.3)$$

$$\bigwedge_{\substack{ov_k,ov_l \in r|_{mv_0} \text{ mit} \\ ov_k \neq ov_l \wedge \\ ov_k|_{otv} = ov_l|_{otv} \wedge \\ ov_k|_{otv}|_{Keys} = \emptyset}} mfm^0_\mu|_{match_o}(ov_k)|_{oiv} \neq mfm^0_\mu|_{match_o}(ov_l)|_{oiv} \qquad \wedge \qquad (4.4)$$

$$\bigwedge_{\substack{ov_k,ov_l \in r|_{mv_0} \text{ mit} \\ ov_k \neq ov_l \wedge \\ ov_k|_{otv} = ov_l|_{otv} \wedge \\ ov_k|_{otv}|_{Keys} \neq \emptyset}} \left\{ \begin{array}{c} (att, mfm^0_\mu|_{match_o}(ov_k).att) : \\ att \in ov_k|_{otv}|_{Keys}|_n \end{array} \right\} \neq \left\{ \begin{array}{c} (att, mfm^0_\mu|_{match_o}(ov_l).att) : \\ att \in ov_l|_{otv}|_{Keys}|_n \end{array} \right\} \qquad (4.5)$$

❍

Das Gleichungssystem *GL* (4.3) stammt aus Definition 3.5.4 (s. S. 86) und erlaubt es, die Attribut- und Identifikatorwerte der potentiell erzeugten Zielmodellfragmente zu berechnen. Durch die Ungleichungssysteme (4.4) und (4.5) wird zusätzlich festgehalten, dass die Objekte des Quellmodells eine Objektbelegung gemäß Definition 3.1.12 (s. S. 59) bilden.

Das Ungleichungssystem (4.4) legt für alle Objektvariablen gleichen Typs ohne Primärschlüsselattribute fest, dass sich die Identitäten der durch sie gematchten Objekte unterscheidet. Dies entspricht Aussage (3.12) in Definition 3.1.12 (Objektbelegung). Das Ungleichungssystem (4.5) formuliert die gleiche Aussage für Objektvariablen, deren Typ Primärschlüsselattribute enthält. Gemäß Aussage (3.11) in der Definition von Objekten (Def. 3.1.11, S. 58) müssen sich zwei Objekte mit Primärschlüsseln in mindestens einem Primärschlüsselattributwert unterscheiden, um unterschiedliche Identitäten zu haben.

Folglich liefert $UGL_{validSrc}$ für alle Matches in metamodellkonformen Modellen dasselbe Ergebnis wie *GL*.

Satz 4.1.1 (Verifikationstechnik für *createsValidFragments*)
Sei $r \in \mathbb{R}$ eine Regel und *GL* das zur *r* gehörige Gleichungssystem aus Definition 3.5.4. Dann gilt:

(i) *GL* hat höchstens eine Lösung und

(ii) $\forall ov_i, ov_j \in r|_{mv_1}|_{OVB}$ mit $ov_i \neq ov_j, ov_i|otv = ov_j|otv$:
$UGL_{valdidSrc} \wedge \left(mfm^1_\mu|_{match_o}(ov_i)|_{oi} = mfm^1_\mu|_{match_o}(ov_j)|_{oi}\right)$ hat keine Lösung.

$\Rightarrow$ *createsValidFragments*(r) (Beweis s. A.2.1, S. 287) ❑

Generell lässt sich leicht zeigen, dass Satz 4.1.1 auch in abgewandelter Form gilt, falls man in Aussage (ii) $UGL_{validSrc}$ durch *GL* ersetzt. In diesem Fall erzeugt eine Regel sicherlich gültige Modellfragmente, falls eine Regelanwendung generell nie Objekte gleichen Typs mit demselben Identifikator erzeugt. Die zusätzlichen Informationen über die Identifikatoren des Quellmodells, die in den Ungleichungen von $UGL_{validSrc}$ formuliert werden, erlauben es jedoch, für eine weitaus größere Klasse von Regeln die Eigenschaft *createsValidFragments* nachzuweisen.

Beispiel: Gegeben sei das in Abbildung 4.3 dargestellte Beispiel einer Regel *r*, für die im Folgenden untersucht werden soll, ob das Prädikat *createsValidFragments*(r) gilt.

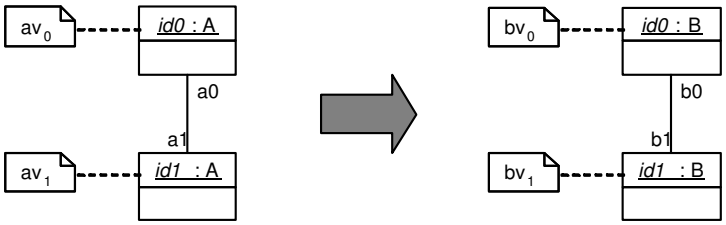


Abbildung 4.3: Die Regel *r*

Das Gleichungssystem *GL* hat demnach für die Regel die folgende Gestalt:

$$\begin{aligned} GL^0_{id}: \; mfm^0_\mu|_{match_o}(av_0)|_{oi} &= id0 \\ mfm^0_\mu|_{match_o}(av_1)|_{oi} &= id1 \\ GL^1_{id}: \quad id0 &= mfm^1_\mu|_{match_o}(bv_0)|_{oi} \\ id1 &= mfm^1_\mu|_{match_o}(bv_1)|_{oi} \end{aligned}$$

GL hat genau eine Lösung für die Identifikatoren der Zielmodellvariablen, womit Forderung (i) von Satz 4.1.1 erfüllt ist:

$$\begin{aligned} mfm^1_\mu|_{match_o}(bv_0)|_{oi} &= mfm^0_\mu|_{match_o}(av_0)|_{oi} \\ mfm^1_\mu|_{match_o}(bv_1)|_{oi} &= mfm^0_\mu|_{match_o}(av_1)|_{oi} \end{aligned}$$

Gemäß Definition 4.1.4 ergibt sich für das Ungleichungssystem $UGL_{validSrc}$:

$$\begin{aligned} mfm^0_\mu|_{match_o}(av_0)|_{oi} &= id0 \\ mfm^0_\mu|_{match_o}(av_1)|_{oi} &= id1 \\ id0 &= mfm^1_\mu|_{match_o}(bv_0)|_{oi} \\ id1 &= mfm^1_\mu|_{match_o}(bv_1)|_{oi} \\ mfm^0_\mu|_{match_o}(av_0)|_{oiv} &\neq mfm^0_\mu|_{match_o}(av_1)|_{oiv} \\ &\equiv \\ mfm^1_\mu|_{match_o}(bv_0)|_{oi} &= mfm^0_\mu|_{match_o}(av_0)|_{oi} \\ mfm^1_\mu|_{match_o}(bv_1)|_{oi} &= mfm^0_\mu|_{match_o}(av_1)|_{oi} \\ mfm^1_\mu|_{match_o}(bv_0)|_{oi} &\neq mfm^1_\mu|_{match_o}(bv_1)|_{oi} \end{aligned} \tag{4.6}$$

Offensichtlich hat das Gleichungssystem

$$\begin{aligned} & UGL_{validSrc} \wedge \\ mfm^1_\mu|_{match_o}(bv_0)|_{oi} \quad &= \quad mfm^1_\mu|_{match_o}(bv_1)|_{oi} \end{aligned} \tag{4.7}$$

wie in Satz 4.1.1 (ii) gefordert, keine Lösung, da die Ungleichung (4.6) aus $UGL_{validSrc}$ genau die Negation der Gleichung (4.7) ist.

Somit wurde formal nachgewiesen, dass die Regel *r* innerhalb einer Modellfragmenttransformation niemals denselben Identifikatorwert für die Objektvariablen b_0 und b_1 erzeugen kann, was in diesem einfachen Fall bereits intuitiv erfassbar ist. Dennoch werden für den Nachweis die zusätzlichen in $UGL_{validSrc}$ formulierten Eigenschaften gültiger Quellmodelle benötigt. ❍

Im folgenden Satz wird eine einfachere, wenn auch weniger mächtige, Verifikationstechnik für *createsValidFragments* vorgestellt, die sich nicht auf die in $UGL_{validSrc}$ spezifizierten Constraints über das Quellmodell einer Modellfragmenttransformation stützt.

Satz 4.1.2 (Einfache Verifikationstechnik für *createsValidFragments*)
Sei $r \in \mathbb{R}$ eine Regel und *GL* das zu *r* gehörige Gleichungssystem aus Definition 3.5.4. Dann gilt:

(i) *GL* hat höchstens eine Lösung und

(ii) $\forall ov_i, ov_j \in r|_{mv_1}|_{OVB}$ mit $ov_i \neq ov_j : ov_i \not\sim ov_j$

$\Rightarrow createsValidFragments(r)$ (Beweis s. A.2.1, S. 288) ❑

Der Vorteil dieser vereinfachten Technik ist es, dass sie deutlich intuitiver anwendbar ist und daher in der Praxis bereits beim Erstellen von Regeln berücksichtigt werden kann. Zudem erfordert sie weniger Aufwand bei der Realisierung einer Werkzeuguntertstützung für den automatischen Nachweis dieser Eigenschaft, da hier lediglich Gleichungssysteme und keine Ungleichungssysteme verarbeitet werden müssen. Aus diesem Grund wird die einfache Verifikationstechnik auch in der aktuellen Version der Werkzeugunterstützung verwendet (siehe Kapitel 6).

4.1.2 Deterministische Objektvariable

Eine Objektvariable einer Zielmodellvariablen wird als *deterministisch* bezeichnet, falls alle Objekte mit gleicher Identität, die im Verlauf einer Regelanwendung aus ihr erzeugt werden, auch die gleichen Attributwerte haben.

Definition 4.1.5 (Deterministische Objektvariable *deterministic*(r, ov))
Sei $r \in \mathbb{R}$ eine Regel. Eine Objektvariable der rechten Regelseite $ov \in r|_{mv_1}|_{OVB}$ heißt *deterministisch*, falls die nachfolgende Relation *deterministic* gilt:

$$
\begin{aligned}
&deterministic : \mathbb{R} \times \mathbb{OV} \rightarrow \mathbb{B} \\
&deterministic(r, ov) :\Leftrightarrow \\
&\quad \forall m \in \mathbb{M}_{r|_{mv_0}|_{mm}}, o_i \in \mathbb{O}, o_j \in \mathbb{O} \text{ mit } createsObj(o_i, ov, r_i, m) \wedge createsObj(o_j, ov, r_i, m) : \\
&\qquad o_i|_{oi} = o_j|_{oi} \Rightarrow o_i|_V = o_j|_V
\end{aligned}
$$ ❍

Definition 4.1.5 hält fest, dass eine Objektvariable *ov* einer Regel genau dann deterministisch ist, falls gilt: Für beliebige Quellmodelle *m* gilt für alle aus *ov* erzeugten Objekte mit gleichem Identifikator, dass diese Objekte auch dieselben Attributwerte haben.

Beispiel: Abbildung 4.4 stellt eine Regel dar, deren einzige Zielobjektvariable nicht deterministisch ist. Die Zielobjektvariable vom Typ Person hat die Variable *o* als Wert des Primärschlüsselattributs personID. Somit wird die Raumnummer des Quellmodells als Personalnummer im Zielmodell verwendet. Umgekehrt wird in diesem Beispiel die Personalnummer von Objekten des Typs Employee im Quellmodell in das Attribut room kopiert.

Durch diesen Fehler werden von der Regel für ein Office-Objekt im Quellmodell, welches mit zwei Employee-Objekten verbunden ist, zwei neue Objekte des Typs Person erzeugt, deren Attribut personID in beiden Fällen dieselbe Raumnummer *o* des Quellmodells enthält. Da personID das einzige Primärschlüsselattribut der Klasse Person ist, bedeutet dies, dass beide Objekte dieselbe Identität haben.

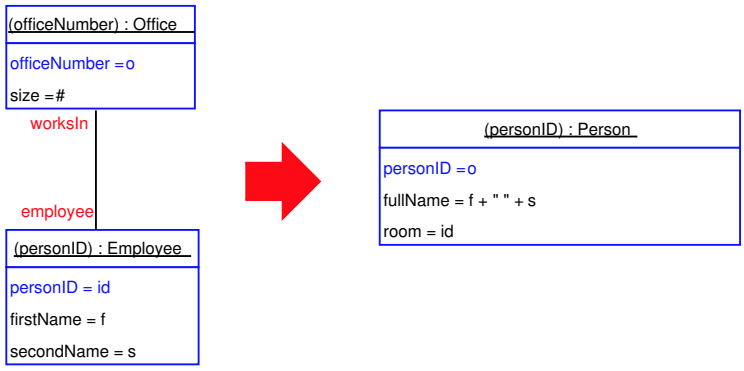


Abbildung 4.4: Beispiel für eine *nicht* deterministische Objektvariable vom Typ Person

Außer in dem unwahrscheinlichen Fall, dass beide Employee-Objekte denselben Vor- und Nachnamen aufweisen, entsteht so ein Konflikt durch widersprüchliche Werte für das Attribut fullName des erzeugten Person-Objektes.

Durch zwei Modellfragmenttransformationen wird also von der Zielobjektvariablen zweimal ein Objekt mit derselben Identität erzeugt, dessen Attributwerte sich jedoch unterscheiden. D.h. die Objektvariable ist nicht deterministisch gemäß Definition 4.1.5. ❍

Im Folgenden werden eine Reihe von Hilfsdefinitionen eingeführt, welche für die Verifikationstechniken zum Nachweis, dass eine Objektvariable deterministisch ist, benötigt werden. Die Abbildung *dependsOn* liefert zu einer Objektvariablen *ov* einer rechten Regelseite alle Objektvariablen der linken Regelseite, von deren Identität die Identität der von *ov* erzeugten Objekte abhängt.

Die nachstehend eingeführte Abbildung *dependsOn* liefert die minimale Menge von Quellobjektvariablen aus denen sich die Identität eines Zielobjekts errechnen lässt. Die Identität des durch eine Objektvariable erzeugten Objekts muss sich also ausschließlich aus Primärschlüsselattributen oder Identifikatoren der Quellobjektvariablen berechnen lassen.

Definition 4.1.6 (*dependsOn*(ov,r))
dependsOn ist eine Abbildung, so dass gilt:

$$dependsOn : \mathbb{OV} \times \mathbb{R} \rightarrow \mathscr{P}(\mathbb{OV}) \cup \{\perp, \diamondsuit\}$$

$$dependsOn(ov,r) \mapsto \begin{cases} \emptyset & \text{falls } ov \in r|_{mv_1}|_{OVB} \text{ und } ov|_{oiv} \text{ bzw. alle Schlüsselattribute von ‚}ov\text{‘ konstant sind.} \\ OV \subseteq r|_{mv_0}|_{OVB} & \text{falls } ov \in r|_{mv_1}|_{OVB} \text{ und } ov|_{oiv} \text{ bzw. alle Primärschlüsselattribute von } ov \text{ eins-zu-eins abhängig zu den Identifikatoren bzw. Primärschlüsselattributen der Elemente aus } OV \text{ sind.} \\ \diamondsuit & \text{falls } ov \in r|_{mv_1}|_{OVB} \text{ und } ov|_{oiv} = \diamondsuit \text{ gilt} \\ \perp & \text{sonst} \end{cases}$$ ❍

Für eine Regel $r \in \mathbb{R}$ und eine Zielobjektvariable $ov \in r|_{mv_1}|$ liefert $dependsOn(ov,r)$ also die minimale Menge von Objektvariablen $\{ov_0,...,ov_m\} \subseteq r_i|_{mv_0}|_{OVB}$ der linken Regelseite, von denen der Identifikator bzw. die Primärschlüsselattribute von *ov* deterministisch berechnet werden können. Falls $ov|_{oiv}$ einen konstanten Wert hat, so liefert $dependsOn(ov,r_i)$ die leere Menge $\emptyset$.

Analog zur *dependsOn*-Abbildung liefert die Abbildung *attDependsOn* die Menge von Quellobjektvariablen einer Regel, von denen der Wert eines Attributs in einer Zielobjektvariablen abhängt.

Definition 4.1.7 (*attDependsOn*(ov, att, r))
Die Abbildung *attDependsOn* ist in folgender Weise definiert:

$$attDependsOn : \mathbb{OV} \times (\mathbb{ID} \times Term_{TA}) \times \mathbb{R} \to \mathscr{P}(\mathbb{OV}) \cup \{\bot\}$$

$$attDependsOn(ov, att, r_i) = \begin{cases} \emptyset & \text{falls } ov \in r|_{mv_1}|_{OVB} \wedge att \in ov|_{VV}|_a \wedge ov.att \in \mathbb{T}|_{T_{\text{val}}} \cup \{\diamondsuit\} \\ OV \subseteq mv_0|_{OVB} & \text{falls } ov \in r|_{mv_1}|_{OVB}, att \in ov|_{VV}|_a \text{ und } att \text{ genau aus den Attributtermen der Objektvariablen in } OV \text{ berechnet werden kann} \\ \bot & \text{sonst} \end{cases}$$ ❍

Die Relation $\overset{mv_0}{\rightsquigarrow}$ bestimmt für eine Quellmodellvariable mv_0, ob eine Menge von Objektvariablen genügt, um bereits die Identität der von einer zweiten Menge von Objektvariablen gematchten Objekte in einem beliebigen Quellmodell festzulegen.

Definition 4.1.8 (*determines*: ($OV^1 \overset{mv_0}{\rightsquigarrow} OV^2$))
Sei $mv_0 \in \mathbb{MV}$ eine Modellvariable und $OV^1, OV^2 \subseteq mv_0|_{OVB}$. Die Relation $OV^1 \overset{mv_0}{\rightsquigarrow} OV^2$ (sprich: OV^1 „determines“ OV^2 in Modellvariable mv_0) ist dann folgendermaßen definiert:

$$\rightsquigarrow : \mathbb{OV} \times \mathbb{MV} \times \mathbb{OV} \to \mathbb{B}$$

$$OV^1 \overset{mv_0}{\rightsquigarrow} OV^2 :\Leftrightarrow$$

$$\forall m \in \mathbb{M}_{mv_0|_{mm}}, mfm_i \in MFM(m, mv_0), mfm_j \in MFM(m, mv_0) :$$

$$\left(\forall ov_1 \in OV^1 : mfm_i(ov_1) = mfm_j(ov_1) \Rightarrow \forall ov_2 \in OV^2 : mfm_i(ov_2) = mfm_j(ov_2)\right)$$ ❍

Demzufolge bedeutet $OV^1 \overset{mv_0}{\rightsquigarrow} OV^2$: Wenn die Objektvariablen in OV^1 auf dieselben Objekte des Quellmodells matchen, dann matchen auch alle Objektvariablen in OV^2 auf dieselben Objekte des Quellmodells.

Zwei wesentliche Eigenschaften der $\rightsquigarrow$-Relation sind ihre Reflexivität und Transitivität. Diese Eigenschaften werden innerhalb der folgenden beiden Lemmas festgehalten:

Lemma 4.1.2 ($\rightsquigarrow$ ist reflexiv)
Sei $mv_0 \in \mathbb{MV}$ eine Modellvariable und $OV \subseteq mv_0|_{OVB}$ eine Menge von Objektvariablen. Dann gilt:

$$OV \overset{mv_0}{\rightsquigarrow} OV$$

D.h. $\rightsquigarrow$ ist reflexiv. (Beweis s. A.2.1, S. 288) ❑

Lemma 4.1.3 ($\rightsquigarrow$ ist transitiv)
Sei $mv_0 \in \mathbb{MV}$ eine Modellvariable und $OV^1, OV^2, OV^3 \subseteq mv_0|_{OVB}$ Mengen von Objektvariablen. Dann gilt immer:

$$OV^1 \overset{mv_0}{\rightsquigarrow} OV^2 \wedge OV^2 \overset{mv_0}{\rightsquigarrow} OV^3 \Rightarrow OV^1 \overset{mv_0}{\rightsquigarrow} OV^3$$

D.h. $\rightsquigarrow$ ist transitiv. (Beweis s. A.2.1, S. 288) ❑

Im allgemeinen Fall ist eine Aussage der Art $OV^1 \overset{mv_0}{\rightsquigarrow} OV^2$ nicht immer leicht nachzuweisen. Insbesondere Modellvariablen mit zyklischen Strukturen und festen Multiplizitätsobergrenzen erfordern oftmals nicht-triviale Techniken zum Nachweis, dass die $\rightsquigarrow$-Relation in bestimmten Fällen gilt. Die im Folgenden vorgestellte Verifikationstechnik für lediglich zwei Objektvariable reicht jedoch für die allermeisten praktischen Anwendungen aus und ist darüber hinaus im Rahmen einer Werkzeugunterstützung vergleichsweise einfach realisierbar.

Satz 4.1.3 (Verifikationstechnik für $\rightsquigarrow$ bei zwei Objektvariablen)
Sei $mv_0 \in \mathbb{MV}$ eine beliebige aber feste Modellvariable und $OV^1, OV^2 \subseteq mv_0|_{OVB}$ Mengen von Objektvariablen. Für $ov_1 \in OV^1$ und $ov_2 \in OV^2$ gilt $ov_1 \overset{mv_0}{\rightsquigarrow} ov_2$, wenn eine Folge $\big((ovae_0, \overline{ovae}_0), \ldots, (ovae_n, \overline{ovae}_n)\big)$ existiert, mit

$$\forall i \in \{0,\ldots,n\} : \{ovae_i, \overline{ovae}_i\} \in mv_0|_{OVAE} \tag{4.8}$$
$$\wedge\ ovae_0|_{ov} = ov_1 \tag{4.9}$$
$$\wedge\ \overline{ovae}_n|_{ov} = ov_2 \tag{4.10}$$
$$\wedge\ \forall i \in \{0,\ldots,n-1\} : \overline{ovae}_i|_{ov} = ovae_{i+1}|_{ov} \tag{4.11}$$
$$\wedge\ \forall i \in \{0,\ldots,n\} : \overline{ovae}_i|_{ae}|_m \in \{(0,1),(1,1)\} \tag{4.12}$$

(Beweis s. A.2.1, S. 288) ❑

Der Satz formuliert genau den Fall, in dem die Objektvariable ov_2 über eine Kette von „Zu-Eins-Assoziationen“ von der Objektvariablen ov_1 aus erreichbar ist. Dieser Umstand wird durch die einzelnen Aussagen des Satzes festgehalten:

- (4.8) legt fest, dass jedes Tupel $(ovae_i, \overline{ovae}_i)$ der Folge eine Objektvariablenassoziation in mv_0 ist.
- (4.9) bestimmt, dass die erste Objektvariablenassoziation der Folge von ov_1 ausgeht.
- (4.10) bestimmt, dass die letzte Objektvariablenassoziation der Folge in ov_2 endet.
- (4.11) stellt sicher, dass die Folge von Objektvariablenassoziationen zusammenhängend ist.
- (4.12) legt schließlich fest, dass alle ausgehenden Assoziationsmultiplizitäten auf dem Pfad ov_1 zu ov_2 höchstens die Obergrenze 1 haben.

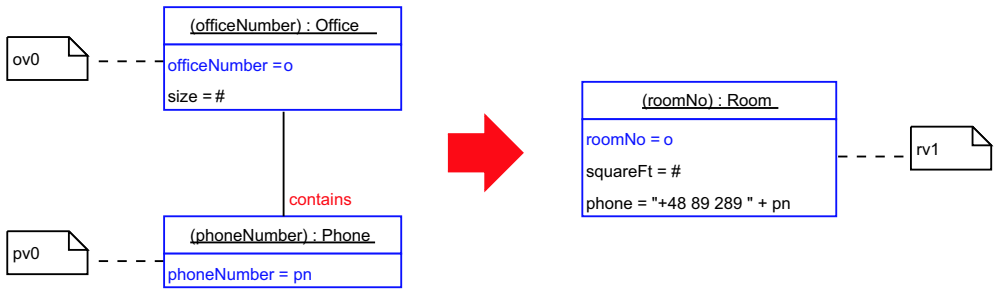


Abbildung 4.5: Die Regel r_1, für deren Quellmodellvariable $r_1|_{mv_0}$ gilt: $\{ov0\} \overset{r_1|_{mv_0}}{\rightsquigarrow} \{pv0\}$

Beispiel: Die Regel aus Abbildung 4.5 zeigt eine Quellmodellvariable, die aus zwei Objektvariablen $ov0$ und $pv0$ besteht. Das Metamodell dieser Modellvariable ist das Metamodell mm_α aus Abbil-

dung 3.5 auf Seite 57. In diesem Beispiel legt $ov0$ die Identität von $op0$ fest, da gemäß mm_α jedes Phone-Objekt zu höchstens *einem* Office-Objekt gehören darf. Also gilt: $\{ov0\} \overset{r_1|_{mv_0}}{\rightsquigarrow} \{pv0\}$.

Auch trivialere determines-Relationen gelten, wie z.B.: $\{ov0, pv0\} \overset{r_1|_{mv_0}}{\rightsquigarrow} \{pv0\}$. ❍

Der nachfolgende Satz bietet eine Verifikationstechnik für den Determinismus einer Objektvariable, welche auf den Abbildungen *dependsOn*, *attDependsOn* und der $\rightsquigarrow$-Relation basiert. Informell besagt der Satz, dass eine Objektvariable *ov* deterministisch ist, falls durch die Menge der Quellobjekte, aus denen sich die Identität eines von *ov* erzeugten Objekts berechnet, auch die Menge aller gematchten Objekte, aus denen sich die Werte der erzeugten Attribute errechnen, bestimmt wird.

Satz 4.1.4 (Verifikationstechnik für den Determinismus einer Objektvariablen)
Sei $r \in \mathbb{R}$ eine Regel und $ov \in r|_{mv_1}|_{OVB}$ eine Objektvariable der rechten Regelseite. Dann gilt:

$$ov|_{oiv} = \diamondsuit \tag{4.13}$$

$$\vee\ \forall (a,t) \in ov|_{VV} \text{ mit } a \notin ov|_{otv}|_{Keys} : t \in \diamondsuit \cup \{t|_{T_{\text{val}}} : (a,t) \in ov|_{otv}|_A\} \tag{4.14}$$

$$\vee\ \big(dependsOn(ov,r) \notin \{\bot, \diamondsuit\} \tag{4.15}$$

$$\wedge \forall a \in ov|_{VV}|_a \text{ mit } a \notin ov|_{otv}|_{Keys} : dependsOn(ov,r) \overset{r|_{mv_0}}{\rightsquigarrow} attDependsOn(ov,a,r)\big)$$

$$\Longrightarrow deterministic(r,ov) \qquad \text{(Beweis s. A.2.1, S. 289)}\ ❑$$

Beispiel: Im Folgenden wird nachgewiesen, dass die Zielobjektvariable $rv1$ der Regel r_1 aus Abbildung 4.5 deterministisch ist. Offensichtlich hängt die Identität der Zielobjektvariablen ausschließlich von $ov0$ ab, d.h. es gilt:

$$dependsOn(rv1, r_1) = \{ov0\}$$

Der Wert des Attributes squareFt ist $\diamondsuit$, während sich der Wert des Attributes phone ausschließlich aus dem einzigen Attribut phoneNumber der Objektvariablen $pv0$ errechnet. Dementsprechend gilt:

$$attDependsOn(rv_1, \text{squareFt}, r_1) = \emptyset$$
$$attDependsOn(rv_1, \text{phone}, r_1) = \{pv0\}$$

Nun lässt sich der Determinismus von $rv1$ anhand von Satz 4.1.4 (4.15) nachweisen:

$$dependsOn(rv1, r_1) = \{ov0\} \overset{r_1|_{mv_0}}{\rightsquigarrow} \emptyset = attDependsOn(rv_1, \text{squareFt}, r_1) \tag{4.16}$$

$$dependsOn(rv1, r_1) = \{ov0\} \overset{r_1|_{mv_0}}{\rightsquigarrow} \{pv0\} = attDependsOn(rv_1, \text{phone}, r_1) \tag{4.17}$$

(4.16) gilt trivialer Weise, die Gültigkeit der $\rightsquigarrow$-Relation in (4.17) wurde bereits im vorangegangen Beispiel (siehe S. 118) gezeigt. Somit ist die Zielobjektvariable $rv1$ deterministisch. ❍

4.1.3 Konfliktfreie Objektvariable

Innerhalb dieses Abschnitts werden Mechanismen vorgestellt, die es erlauben, Aussagen darüber zu machen, ob zwei unterschiedliche Zielobjektvariablen bei der Anwendung eines Regelwerks Konflikte erzeugen können. Dies ist der Fall, falls beide Objektvariablen ein Objekt mit der gleichen Identität und widersprüchlichen Attributwerten erzeugen. Im Gegensatz zum Determinismus von Objektvariablen entstehen die Konflikte durch widersprüchliche Attributwerte hier jedoch durch das Erzeugen eines Objektes aus *verschiedenen* Objektvariablen gleichen Typs.

Definition 4.1.9 (Konfliktfreie Objektvariable $conflictFree(R, ov_i, ov_j)$)
Seien $r_i, r_j \in R \in \mathbb{RW}$ zwei (nicht notwendigerweise verschiedene) Regeln eines Regelwerks. Zwei Objektvariablen der jeweils rechten Regelseite $ov_i \in r_i|_{mv_1}|_{OVB}$ und $ov_j \in r_j|_{mv_1}|_{OVB}$ heißen *konfliktfrei*, falls die Relation $conflictFree(R, ov_i, ov_j)$ für sie gilt:

$$conflictFree : \mathbb{RW} \times \mathbb{OV} \times \mathbb{OV} \to \mathbb{B}$$
$$conflictFree(R, ov_i, ov_j) :\Leftrightarrow$$
$$\forall M \in M_R^0, o_i, o_j \text{ mit } createsObj(o_i, ov_i, r_i, srcModel(M, r_i)) \wedge$$
$$createsObj(o_j, ov_j, r_j, srcModel(M, r_j)) \; : mergeable(\{o_i\}, \{o_j\})$$ ❍

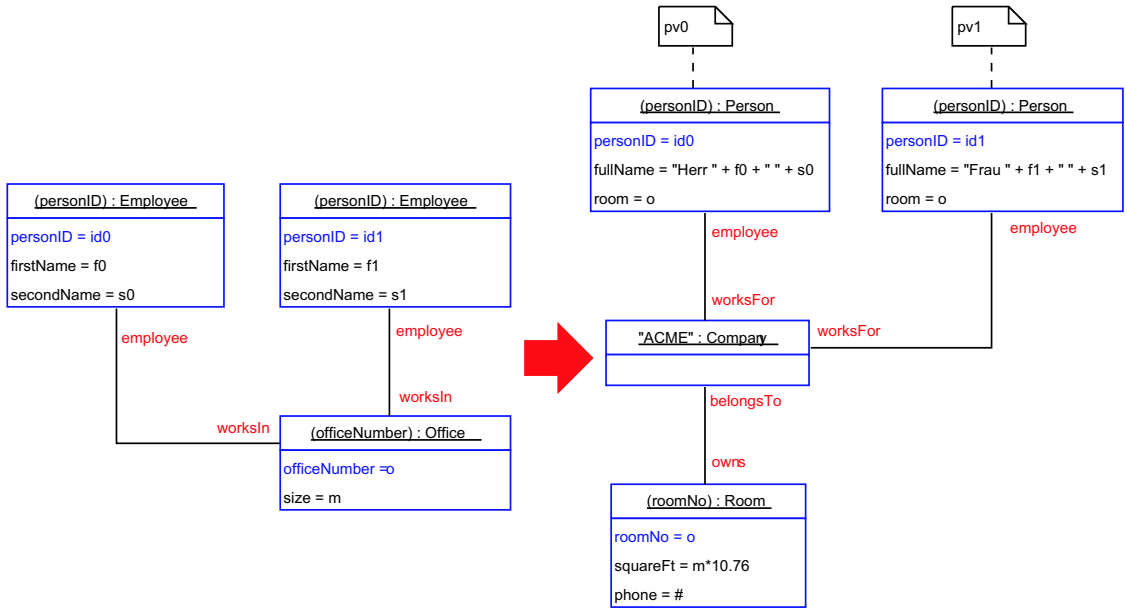


Abbildung 4.6: Beispiel für eine Regel mit zwei *nicht* konfliktfreien Objektvariablen *pv*0 und *pv*1

Beispiel: Abbildung 4.6 zeigt eine Regel, welche für alle Objekte des Typs Office die mit zwei Objekten des Typs Employee verbunden sind, ein Zielmodellfragment erzeugt, das wiederum zwei Objekte des Typs Person enthält. Der Attributwert für den Namen berechnet sich hierbei jeweils aus dem Vor- und Nachnamen der gefundenen Employee-Objekte, jedoch wird dem Namen jeweils einmal die Zeichenkette Frau und einmal die Zeichenkette Herr vorangestellt.

Falls diese Regel jemals auf ein Modellfragment matcht, so wird dies mit Sicherheit zweimal geschehen. Zuerst matcht beispielsweise ein Employee-Objekt mit der ID „17“ dem Namen „Mustermann“ auf die linke Objektvariable und ein anderes mit ID „18“ und dem Namen „Mustermeier“ auf die rechte Objektvariable. Dies würde zur Erzeugung von zwei Objekten mit der ID „17“ und dem Namen „Herr Mustermann“ bzw. „18“ und „Frau Mustermeier“ durch die Objektvariablen *pv*0 bzw. *pv*1 führen.

Ein zweites Mal würde die linke Regelseite dasselbe Modellfragment in umgekehrter Weise matchen. Nun werden zwei Objekte mit ID „18“ und Namen „Herr Mustermeier“ sowie ID „17“ „Frau Mustermann“ erzeugt. Abbildung 4.7 stellt die beiden erzeugten Modellfragmente dar. Offensichtlich können diese beiden Modellfragmente nicht zu einem Zielmodellfragment zusammengeführt werden,

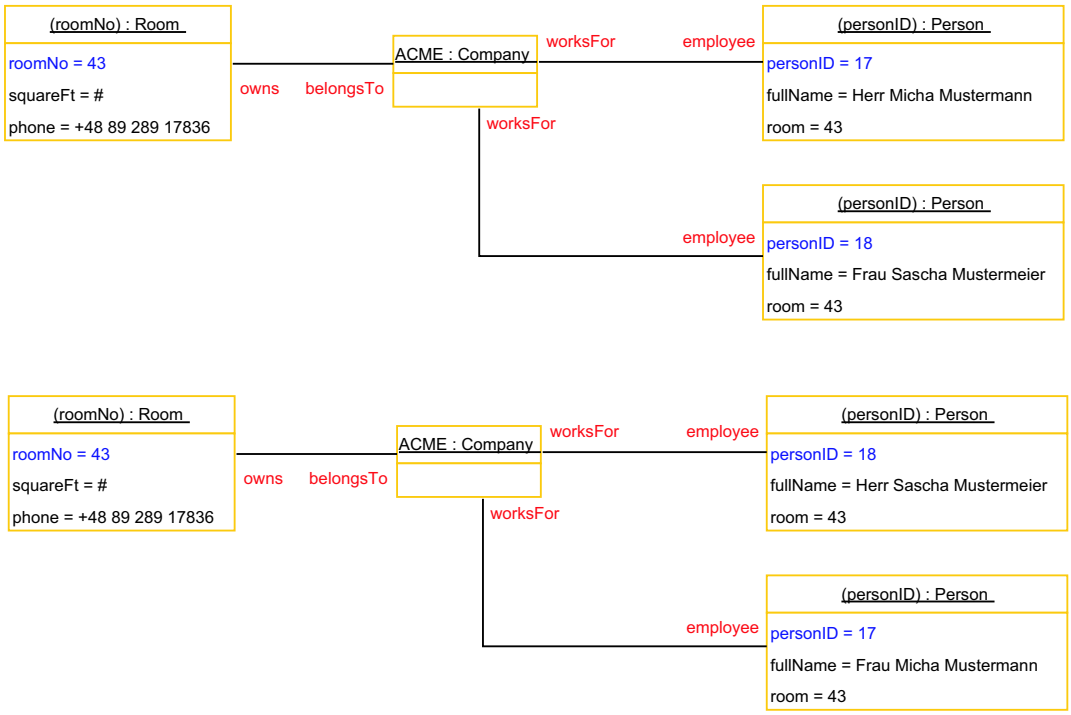


Abbildung 4.7: Zwei von der Regel aus Abbildung 4.6 erzeugte Modellfragmente

da sich die Werte des Attributs fullName widersprechen. Die beiden Objektvariablen $pv0$ und $pv1$ sind folglich *nicht* konfliktfrei, da durch sie erzeugte Objekte zu Konflikten im Zielmodellfragment führen können. ❍

Es wird nun die Relation $potConflicting$ eingeführt. Diese gilt genau dann, falls zwei Zielobjektvariablen eines Regelwerks für eine beliebige aber feste Menge von Quellmodellen das selbe Objekt im Zielmodell erzeugen können.

Definition 4.1.10 (Konfliktgefährdet $potConflicting(R, ov_i, ov_j)$)
Seien $r_i, r_j \in R \in \mathbb{RW}$ zwei (nicht notwendiger Weise verschiedene) Regeln eines Regelwerks mit $ov_i \in r_i|_{OVB}$, $ov_j \in r_j|_{OVB}$ und $ov_i \neq ov_j$. ov_i und ov_j heißen *konfliktgefährdet*, falls die im Folgenden definierte Relation $potConflicting$ gilt:

$$potConflicting : \mathbb{RW} \times \mathbb{OV} \times \mathbb{OV} \rightarrow \mathbb{B}$$
$$potConflicting(R, ov_i, ov_j) :\Leftrightarrow$$
$$\exists M \in M_R^0, o_i, o_j \text{ mit } createsObj(o_i, ov_i, r_i, srcModel(M, r_i)) \wedge$$
$$createsObj(o_j, ov_j, r_j, srcModel(M, r_j)) \;:\; o_i|_{oi} = o_j|_{oi} \wedge o_i|_{ot} = o_j|_{ot}$$ ❍

Das nachstehende Lemma hält fest, dass zwei Objektvariablen, die nicht konfliktgefährdet sind auch konfliktfrei sind. Dieser Sachverhalt ist offensichtlich, da durch Objektvariable, die niemals das selbe Objekt erzeugen können, auch keine Konflikte in den Attributwerten eines solchen Objektes entstehen können.

Lemma 4.1.4 (Konfliktfreiheit von Objektvariablen)
Für $R \in \mathbb{RW}$ und $ov_i, ov_j \in R|_{mv_1}|_{OVB}$ mit $ov_i \neq ov_j$ gilt:

$$\neg potConflicting(R, ov_i, ov_j) \Rightarrow conflictFree(R, ov_i, ov_j) \qquad \text{(Beweis s. A.2.1, S. 289)} \ \square$$

Ein einfacher Weg um festzustellen, dass zwei Objektvariablen offensichtlich nicht konfliktgefährdet sind, ist es sie auf ihre Ähnlichkeit hin zu vergleichen (siehe Definition 3.5.8, S. 91). Lemma 4.1.5 hält dies fest:

Lemma 4.1.5 (Einfache Verifikationstechnik für $\neg potConflicting$)
Sei $R \in \mathbb{RW}$ ein Regelwerk mit $r_i, r_j \in R$. Für Objektvariablen $ov_i, ov_j \in R|_{mv_1}|_{OVB}$ gilt:

$$ov_i \not\sim ov_j \tag{4.18}$$
$$\Rightarrow \neg potConflicting(R, ov_i, ov_j) \qquad \text{(Beweis s. A.2.1, S. 290)} \ \square$$

Lemma 4.1.5 ermöglicht es, die Konfliktfreiheit zweier Objektvariablen auf einfache Weise nachzuweisen. Diese einfache Verifikationstechnik wird im nachfolgende Lemma verwendet. Zudem überprüft das Lemma, ob die Attribute zweier Objektvariablen entweder paarweise identische konstante Werte, den Wert Diamond oder aber denselben Term innerhalb der selben Regel enthalten. In allen drei Fällen können Konflikte zwischen den Attributen ausgeschlossen werden.

Lemma 4.1.6 (Einfache Verifikationstechnik für Konfliktfreiheit)
Sei $R \in \mathbb{RW}$ ein Regelwerk. Dann gilt für zwei Zielobjektvariablen $ov_j, ov_k \in R|_{mv_1}|_{OVB}$ des Regelwerks mit $ov_k \neq ov_j$:

$$\neg potConfilicting(R, ov_j, ov_k) \vee \tag{4.19}$$
$$\forall n \in \{n : \exists ov_k|_{otv}|_A \ni (n,t) \notin ov_k|_{otv}|_{Keys}\} : \tag{4.20}$$
$$\big(\ \diamondsuit \in ov_j.n \cup ov_k.n$$
$$\vee\, ov_j.n = ov_k.n \wedge r_j = r_k$$
$$\vee\, ov_j.n = ov_k.n \wedge ov_j.n, ov_k.n \text{ sind konstante Werte } \big)$$
$$\Longrightarrow conflictFree(R, ov_k, ov_j) \qquad \text{(Beweis s. A.2.1, S. 290)} \ \square$$

Der Vorteil der in Lemma 4.1.6 festgehaltenen Verifikationstechnik ist ihre intuitive Anwendbarkeit. Sie ermöglicht es, bereits bei der Spezifikation einer Modelltransformation die Regeln per Augenscheinnahme auf die Konfliktfreiheit von Objektvariablen hin zu überprüfen. Für den Nachweis der Konfliktfreiheit anhand dieser Technik müssen lediglich die betroffenen Zielobjektvariablen untersucht werden.

Im Folgenden wird eine mächtigere Verifikationstechnik für Regeln mit demselben Quellmetamodell vorgestellt, die auch die Quellmodellvariablen der jeweiligen Regeln berücksichtigt. Hierzu wird eine Reihe von Aussagen in Form von Gleichungssystemen und Ungleichungen gemacht, in denen für den Nachweis der Konfliktfreiheit relevante Informationen formuliert werden. Im einzelnen sind dies:

- Zwei unterschiedliche Modellfragmenttransformationen einer Regelwerksanwendung erzeugen das gleiche Objekt (Gleichungssystem *PCONF*).
- Die Objekte eines Quellmodells bilden eine gültige Objektbelegung gemäß Definition 3.1.12 (Aussage *VALIDOB*).

- Alle Attribute der von verschiedenen Objektvariablen erzeugten Objekte haben paarweise identische Attribute oder in mindestens einem Attribut den Wert $\diamondsuit$ (Gleichungssystem *EQATTS*).

Im Verlauf einer Transformation werden Identifikatoren und Attributwerte neu erzeugter Objekte aus den Lösungen des in Definition 3.5.4 aufgestellten Gleichungssystems *GL* errechnet. Die Lösung dieses Gleichungssystems für die Werte eines neuen Objektes (in Abhängigkeit von den in einem Match gefundenen Attribut- und Identifikatorwerten) wird nun als GL' eingeführt. GL' wird im weiteren Verlauf benötigt, um feststellen zu können, ob verschiedene Objektvariable dasselbe Objekt mit unterschiedlichen Attributwerten erzeugen können. Dies würde bedeuten, dass das Regelwerk, zu dem sie gehören, nicht anwendbar wäre.

Definition 4.1.11 ($GL'(r, ov_i, mfm_k^0)$)
Sei $r \in R \in \mathbb{RW}$ eine Regel eines Regelwerks mit $ov_i \in r|_{mv_1}|_{OVB}$. Für einen Modellfragment-Match mfm_k der linken Regelseite von r wird die Lösung des Gleichungssystems aus Definition 3.5.4 mit $GL'(r, ov_i, mfm_k^0)$ bezeichnet. Hierbei enthält $GL'(r, ov_i, mfm_k^0)$ nur die Lösungen der Identifikator- und Attributwerte von ov_i für den Match mfm_k einer beliebigen Match-Folge von r im jeweiligen Quellmodell.

Dementsprechend enthält $GL'(r, ov_i, mfm_k^0)$ also Gleichungen der Form:

$$mfm_k^1|_{match_o}(ov_i)|_{oi} = f_{id}(mfm_k^0|_{mf}|_{OB}) \wedge$$
$$\bigwedge_{att \in ov_i|_{VV}|_a} mfm_k^1|_{match_o}(ov_i).att = f_{att}(mfm_k^0|_{mf}|_{OB})$$ ❍

GL' liefert alle Gleichungen, die im Verlauf einer Regelanwendung für die Bestimmung der Identität und der Attribute, der aus ov_i erzeugten Objekte, notwendig sind. Der Wert k dient dabei als Index für den jeweiligen Match, auf den sich die Berechnung bezieht.

Beispiel: Im Folgenden wird die in Abbildung 4.8 dargestellte Regel r_{appl} mit den zwei Objektvariablen bv_0 und bv_1 gleichen Typs auf der rechten Seite betrachtet.

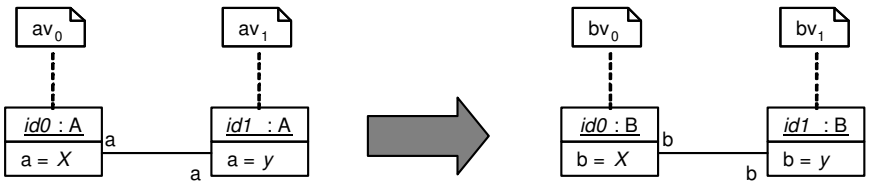


Abbildung 4.8: Regel r_{appl}

Gemäß Definition 3.5.4 erhält man für r_{appl} die folgende Menge von Gleichungen für das Gleichungssystem *GL*.

$$\begin{aligned}
GL_{id}^0 : mfm_h^0|_{match_o}(av_0)|_{oi} &= id0 \\
mfm_h^0|_{match_o}(av_1)|_{oi} &= id1 \\
GL_v^0 : mfm_h^0|_{match_o}(av_0).a &= x \\
mfm_h^0|_{match_o}(av_1).a &= y \\
GL_{id}^1 : id0 &= mfm_h^1|_{match_o}(bv_0)|_{oi} \\
id1 &= mfm_h^1|_{match_o}(bv_1)|_{oi} \\
GL_v^1 : x &= mfm_h^1|_{match_o}(bv_0).b \\
y &= mfm_h^1|_{match_o}(bv_1).b
\end{aligned}$$

Für ein beliebiges aber festes Quellmodell $m \in \mathbb{M}_{r_n appl|_{mv_0}|_{mm}}$ sei $\{match_0^0, \ldots, match_n^0\} := MFM(m, r_{appl}|_{mv_0})$ die Modellfragment-Match-Menge der linken Regelseite von r_{appl} in m. GL gilt für jedes h mit $0 \leq h \leq n$, d.h. es gilt für jeden möglichen Match im Quellmodell.

Das angegebenen Gleichungssystem lässt sich in ein äquivalentes, einfacheres System umformen das die Lösungen für die Attribute und Identifikatoren des Zielmodells aufzeigt. Für das Gleichungssystem $GL'(r_{appl}, bv_0, mfm_k^0)$ ergibt sich gemäß Definition 4.1.11:

$$\begin{aligned} GL'(r_{appl}, bv_0, mfm_k^0) : \; & mfm_k^1|_{match_o}(bv_0)|_{oi} = mfm_k^0|_{match_o}(av_0)|_{oi} \\ & mfm_h^1|_{match_o}(bv_0).b = mfm_h^0|_{match_o}(av_0).a \end{aligned}$$

Analog ergibt sich für $GL'(r_{appl}, bv_1, mfm_l^0)$:

$$\begin{aligned} GL'(r_{appl}, bv_0, mfm_l^0) : \; & mfm_l^1|_{match_o}(bv_1)|_{oi} = mfm_l^0|_{match_o}(av_1)|_{oi} \\ & mfm_l^1|_{match_o}(bv_1).b = mfm_l^0|_{match_o}(av_1).a \end{aligned}$$

❍

Es wird nun das Gleichungssystem *PCONF* definiert, welches formal den Umstand charakterisiert, dass in zwei Zielmodellfragment-Matches durch zwei unterschiedliche Objektvariablen jeweils ein Objekt desselben Typs und mit derselben Identität erzeugt wird. Diese Situation ist von besonderem Interesse, da sie genau den Zustand beschreibt, in dem es zu einem Konflikt zwischen den beiden Objektvariablen kommen kann.

Definition 4.1.12 ($PCONF(R, ov_i, ov_j, mfm_k^0, mfm_l^0)$)
Seien $r_i, r_j \in R \in \mathbb{RW}$ zwei (nicht notwendiger Weise verschiedene) Regeln eines Regelwerks, die gültige Modellfragmente gemäß Definition 4.1.3 erzeugen. Weiter sei $ov_i \in r_i|_{mv_1}|_{OVB}$, $ov_j \in r_j|_{mv_1}|_{OVB}$, $ov_i \neq ov_j$ und $ov_i|_{otv} = ov_j|_{otv}$. Dann ist das Gleichungssystem $PCONF(R, ov_i, ov_j, mfm_k^0, mfm_l^0)$ definiert als:

$$\begin{aligned} PCONF(R, ov_i, ov_j, mfm_k^0, mfm_l^0) : \; & mfm_k^1|_{match_o}(ov_i)|_{oi} = mfm_l^1|_{match_o}(ov_j)|_{oi} \; \wedge \\ & mfm_k^1|_{match_o}(ov_i)|_{ot} = mfm_l^1|_{match_o}(ov_j)|_{ot} \; \wedge \\ & GL'(r_i, ov_i, mfm_k^0) \; \wedge \\ & GL'(r_j, ov_j, mfm_l^0) \end{aligned}$$

Die Unbekannten dieses Gleichungssystems sind alle gematchten Identifikatoren und Attributwerte im Quellmodell. Variablen aus verschiedenen Regeln mit gleichem Namen werden hierbei als unterschiedlich angesehen (z.B. $r_0.var \neq r_1.var$).

mfm_k^0 und mfm_l^0 stellen freie Variable für beliebige, unterschiedliche Quell-Matches der Regeln aus R in einem beliebigen, aber festen Quellmodellen dar. ❍

Im folgenden Lemma wird gezeigt, dass zwei Objektvariablen konfliktgefährdet sind, falls Quellobjekte für die Modellfragmenttransformationen gefunden werden können, so dass das Gleichungssystem *PCONF* eine Lösung besitzt. In diesem Fall kann es sein, dass beide Objektvariablen unabhängig voneinander dasselbe Zielobjekt erzeugen.

Lemma 4.1.7 (Nachweis von $\neg potConflicting$)
Seien $r_i, r_j \in R \in \mathbb{RW}$ zwei (nicht notwendiger Weise verschiedene) Regeln eines Regelwerks, die gültige Modellfragmente gemäß Definition 4.1.3 erzeugen. Weiter sei $ov_i \in r_i|_{mv_1}|_{OVB}$, $ov_j \in r_j|_{mv_1}|_{OVB}$, $ov_i \neq ov_j$ und $ov_i|_{otv} = ov_j|_{otv}$.

Falls für das System $PCONF(R, ov_i, ov_j, mfm_k^0, mfm_l^0)$ *keine* Lösung existiert gilt:

$$\neg potConflicting(R, ov_i, ov_j)$$ (Beweis s. A.2.1, S. 290) ❑

Für zwei Objektvariable wird nun das Attribut-Gleichungssystem *EQATTS* eingeführt. Dieses Gleichungssystem macht formal die Aussage, dass beim Erzeugen zweier Objekte aus zwei verschiedenen Objektvariablen gleichen Typs beide erzeugten Objekte in allen Attributen entweder den gleichen Wert oder den Wert $\diamondsuit$ haben.

Definition 4.1.13 (Attribut-Gleichungssystem ($EQATTS(R, ov_0, ov_1, mfm_k^0, mfm_l^0)$))
Seien $r_i, r_j \in R \in \mathbb{RW}$ zwei Regeln eines Regelwerks mit $ov_i \in r_i|_{mv_1}|_{OVB}$ und $ov_j \in r_j|_{mv_1}|_{OVB}$. ov_i und ov_j seien weiter vom selben Typ, d.h. $ov_i|_{otv} = ov_j|_{otv} =: class$.
Dann bezeichnet $EQATTS(R, ov_0, ov_1, mfm_k^0, mfm_l^0)$ das folgende Gleichungssystem:

$$\bigwedge_{\substack{att \in class|_A|_n \setminus class|_{Keys}|_n \\ \wedge mfm_k^1|_{match_o}(ov_0).att \neq \diamondsuit \\ \wedge mfm_l^1|_{match_o}(ov_1).att \neq \diamondsuit}} mfm_k^1|_{match_o}(ov_0).att = mfm_l^1|_{match_o}(ov_1).att$$

mfm_k^1 und mfm_l^1 sind hierbei als symbolische Variablen aufzufassen, die für Matches der Zielmodellvariablen im Zielmodell stehen. D.h. $mfm_k^1|_{match_o}(ov_0)$ bezeichnet ein durch ov_0 erzeugtes Objekt. ❍

Beispiel: Für die in Abbildung 4.8 dargestellte Regel r_{appl} ergibt sich gemäß Definition 4.1.13 das Gleichungssystem $EQATTS(\{r_{appl}\}, bv_0, bv_1, mfm_k^0, mfm_l^0)$ welches lediglich aus einer einzigen Gleichung besteht:

$$EQATTS : mfm_k^1|match_o(bv_0).b = mfm_l^1|match_o(bv_1).b$$

Die Aussage dieser Gleichung ist, dass im Match mfm_k^0 der Regel r_{appl} durch die Objektvariable bv_0 ein Objekt mit einem Attributwert angelegt wird, welcher der gleiche ist wie der im Match mfm_l^0 durch bv_1 erzeugte. ❍

Mit Hilfe des Attribut-Gleichungssystems und den Lösungen GL' wird nun festgestellt, ob zwei unterschiedliche Objektvariablen einen Konflikt verursachen können, indem sie dasselbe Objekt erzeugen und dabei unterschiedliche Werte für ein Attribut dieses Objektes generieren.

Im folgenden Lemma wird der Fall untersucht, in dem Lösungen von gefundenen Matches im Quellmodell existieren, die dazu führen, dass zwei Objektvariablen dasselbe Objekt erzeugen. Ist jede dieser Lösungen auch eine Lösung für das System *EQATTS*, welches festlegt, dass die Attribute dieser erzeugten Objekte identisch sind, falls sie nicht den Wert $\diamondsuit$ haben, so sind die beiden Objektvariablen konfliktfrei.

Lemma 4.1.8 (Konfliktfreiheit für beliebige Quellstrukturen)
Falls jede Lösung des Gleichungssystems $PCONF(R, ov_i, ov_j, mfm_k^0, mfm_l^0)$ auch eine Lösung für $EQATTS(R, ov_i, ov_j, mfm_k^0, mfm_l^0)$ ist, gilt:

$$conflictFree(R, ov_i, ov_j)$$ (Beweis s. A.2.1, S. 290) ❑

Beispiel: Für das Beispiel aus Abbildung 4.8 ergibt sich das folgende Gleichungssystem:

$$
\begin{aligned}
&PCONF(\{r_{appl}\}, bv_0, bv_1, mfm_k^0, mfm_l^0): \\
&\qquad\qquad mfm_k^1|_{match_o}(bv_0)|_{oi} = mfm_l^1|_{match_o}(bv_1)|_{oi} \\
&GL'(r_{appl}, bv_0, mfm_k^0): \; mfm_k^1|_{match_o}(bv_0)|_{oi} = mfm_k^0|_{match_o}(av_0)|_{oi} \\
&\qquad\qquad mfm_k^1|_{match_o}(bv_0).b = mfm_k^0|_{match_o}(av_0).a \\
&GL'(r_{appl}, bv_0, mfm_l^0): \; mfm_l^1|_{match_o}(bv_1)|_{oi} = mfm_l^0|_{match_o}(av_1)|_{oi} \\
&\qquad\qquad mfm_l^1|_{match_o}(bv_1).b = mfm_l^0|_{match_o}(av_1).a
\end{aligned}
$$

Das Gleichungssystem hat eine Lösung für den Fall

$$mfm_k^0|_{match_o}(av_0)|_{oi} = mfm_l^0|_{match_o}(av_1)|_{oi}$$

Dies bedeutet informell gesehen, dass die Objektvariablen bv_0 und bv_1 dasselbe Objekt erzeugen können, falls auf der linken Seite die Objektvariablen av_0 und av_1 jeweils auf dasselbe Objekt matchen. Folglich sind die beiden Objektvariablen bv_0 und bv_1 konfliktgefährdet.

Nun wird das Gleichungssystem um die Forderung erweitert, dass in diesem Fall sämtliche Attributwerte gleich sein müssen. Dies entspricht dem Hinzufügen des *EQATTS*-Gleichungssystems aus Definition 4.1.13. Insgesamt ergibt sich:

$$
\begin{aligned}
&\qquad\qquad mfm_k^1|_{match_o}(bv_0)|_{oi} = mfm_l^1|_{match_o}(bv_1)|_{oi} \\
&GL'(r_{appl}, bv_0, k): \; mfm_k^1|_{match_o}(bv_0)|_{oi} = mfm_k^0|_{match_o}(av_0)|_{oi} \\
&\qquad\qquad mfm_k^1|_{match_o}(bv_0).b = mfm_k^0|_{match_o}(av_0).a \\
&GL'(r_{appl}, bv_1, l): \; mfm_l^1|_{match_o}(bv_1)|_{oi} = mfm_l^0|_{match_o}(av_1)|_{oi} \\
&\qquad\qquad mfm_l^1|_{match_o}(bv_1).b = mfm_l^0|_{match_o}(av_1).a \\
&EQATTS: \; mfm_k^1|match_o(bv_0).b = mfm_l^1|match_o(bv_1).b
\end{aligned}
$$

Durch das Hinzufügen des Systems *EQATTS* zum Gleichungssystem *PCONF* entsteht ein neuer Constraint für die Lösung:

$$mfm_k^0|_{match_o}(av_0).a = mfm_l^0|_{match_o}(av_1).a \tag{4.21}$$

Der Lösungsraum wird dann nicht weiter eingeschränkt, falls diese Gleichung *immer* wahr ist. Anhand von Lemma 4.1.8 alleine lässt sich die Konfliktfreiheit der beiden Objektvariablen jedoch nicht nachweisen. ❍

Wie auch das Beispiel gezeigt hat, ist der Nachweis der Konfliktfreiheit mit Hilfe von Lemma 4.1.8 nicht immer ohne weiteres möglich, da das Gleichungssystem *EQATTS* die Lösung von *PCONF* in der Regel weiter einschränkt. Der Nachweis ist jedoch in vielen Fällen dann möglich, falls die Lösungen von *PCONF* von vornherein sinnvoll eingeschränkt werden können. Eine solche Einschränkung ist durch die Tatsache gegeben, dass alle Objekte der Quellmodelle konsistente Objektbelegungen gemäß Definition 3.1.12 sind. Dieser Sachverhalt wird durch die Aussage *VALIDOB* formuliert und kann für Objektvariable, deren Regeln dasselbe Quellmetamodell haben, berücksichtigt werden.

Zunächst wird hierzu ein einfaches Lemma eingeführt, welches eine für die Verifikationstechnik der Konfliktfreiheit wichtige Eigenschaft von Objektbelegungen festhält.

Lemma 4.1.9 (Objekte in Objektbelegungen)
Für ein beliebiges Metamodell $mm \in \mathbb{MM}$ und $o_0, o_1 \in OB \in \mathbb{OB}_{mm}$ mit $o_0|_{ot} = o_1|_{ot} =: class$ gilt:

$$\forall att \in class|_A|_n : o_0|_{oi} = o_1|_{oi} \Rightarrow o_0.att = o_1.att$$ (Beweis s. A.2.1, S. 292) ❑

Definition 4.1.14 (*VALIDOB*$(R, mm_0, mfm_k^0, mfm_l^0)$)
Es sei $R' := \{r \in R : r|_{mv_0}|_{mm} = mm_0\}$ die Menge aller Regeln aus $R \in \mathbb{RW}$ die $mm_0 \in \mathbb{MM}$ als Quellmetamodell haben.

Dann ist die Aussage $VALIDOB(R, ov_i, ov_j, mfm_k^0, mfm_l^0)$ in folgender Weise definiert:

$$\bigwedge_{\substack{ov_i, ov_j \in R'|_{mv_0}|_{OVB} \\ \text{mit } ov_i \neq ov_j \wedge ov_i|_{otv} = ov_j|_{otv}}} \Big(mfm_k^0|_{match_o}(ov_i)|_{oi} \neq mfm_l^0|_{match_o}(ov_j)|_{oi} \vee \bigwedge_{att \in class|_A|_n} mfm_k^0|_{match_o}(ov_i).att = mfm_l^0|_{match_o}(ov_j).att \Big)$$

Hierbei stellen mfm_k^0 und mfm_l^0 freie Variable für beliebige, unterschiedliche Quell-Matches der Regeln aus R' in einem beliebigen, aber festen Quellmodell $m \in \mathbb{M}_{mm_0}$ dar. ❍

Das nachfolgende Lemma macht sich die Aussage von Lemma 4.1.9 zunutze. Es hält formal fest, dass zwei Objekte gleichen Typs, die im selben Quellmodell gematcht werden, entweder verschiedene Identitäten haben oder aber sämtliche Attribute paarweise denselben Wert aufweisen.

Lemma 4.1.10 (Gematchte Quellobjekte)
Sei $R \in \mathbb{RW}$ ein Regelwerk mit $r_i, r_j \in R$ und $ov_i \in r_i|_{mv_0}|_{OVB}, ov_j \in r_j|_{mv_0}|_{OVB}$ mit $ov_i|_{otv} = ov_j|_{otv} = class$. Weiterhin haben die Regeln r_i und r_j dasselbe Quellmetamodell $mm_0 := r_i|_{mv_0}|_{mm} = r_j|_{mv_0}|_{mm}$.

Dann gilt $VALIDOB(R, mm_0, mfm_k^0, mfm_l^0)$ für jede Wahl von verschiedenen Matches mfm_k^0 und mfm_l^0 von Regeln in einem gültigen Quellmodell mit dem Quellmetamodell mm_0 (Beweis s. A.2.1, S. 292) ❑

Mit Hilfe des nachstehenden Satzes lässt sich die Konfliktfreiheit von Objektvariablen nachweisen, sofern ihre Regeln dasselbe Quellmetamodell haben:

Satz 4.1.5 (Verifikationstechnik für Konfliktfreiheit)
Sei $R \in \mathbb{RW}$ ein Regelwerk, $r_i, r_j \in R$ zwei Regeln und $ov_i \in r_i|_{mv_0}|_{OVB}$, $ov_j \in r_j|_{mv_0}|_{OVB}$ jeweils Quellobjektvariablen der Regeln, so dass gilt:

(i) Die Regeln r_i und r_j haben dasselbe Quellmetamodell $mm_0 \in \mathbb{MM}$:

$$mm_0 := r_i|_{mv_0}|_{mm} = r_j|_{mv_0}|_{mm}$$

(ii) ov_i und ov_j sind vom selben Typ $class \in \mathbb{C}$:

$$class := ov_i|_{otv} = ov_j|_{otv}$$

Betrachtet werden alle Lösungen von $PCONF(R, ov_i, ov_j, mfm_k^0, mfm_l^0)$, für welche das System aus Gleichungen und Ungleichungen $VALIDOB(R, ov_i, ov_j, mfm_k^0, mfm_l^0)$ gilt.
Ist für jede dieser Lösungen auch das Gleichungssystem $EQATTS(R, ov_i, ov_j, mfm_k^0, mfm_l^0)$ erfüllt, so gilt:

$$conflictFree(R, ov_i, ov_j)$$ (Beweis s. A.2.1, S. 292) ❑

Durch den zusätzlichen Constraint *VALIDOB* wird die Menge der Lösungen von *PCONF* weiter eingeschränkt. Allerdings werden durch diese Einschränkung jetzt nur noch Lösungen berücksichtigt, die bei gültigen Quellobjektbelegungen auftreten können. Dementsprechend verringert sich die Anzahl der Lösungen die auch das Gleichungssystem *EQATTS* erfüllen müssen. Die zusätzliche Berücksichtigung der Eigenschaft, dass Quellmodelle immer gültige Objektbelegungen sind, führt zu der größeren Mächtigkeit der in Satz 4.1.5 vorgestellten Verifikationstechnik gegenüber Lemma 4.1.8.

Beispiel: Für das Beispiel ergibt sich für $VALIDOB(\{r_{appl}\}, r_{appl}|_{mv_0}|_{mm}, mfm_k^0, mfm_l^0)$:

$$\begin{aligned} mfm_k^0|_{match_o}(av_0)|_{oi} &\neq mfm_l^0|_{match_o}(av_1)|_{oi} \\ &\vee \\ mfm_k^0|_{match_o}(av_0).a &= mfm_l^0|_{match_o}(av_1).a \end{aligned}$$

In *PCONF* ist bereits festgelegt, dass beide Matches Objekte mit gleichem Identifikator erzeugen. Folglich muss die zweite Aussage von *VALIDOB* wahr sein, da die erste *PCONF* widerspricht. Dementsprechend muss für jede Lösung von $PCONF \wedge VALIDOB$ gelten:

$$\begin{aligned} mfm_k^0|_{match_o}(av_0)|_{oi} &= mfm_l^0|_{match_o}(av_1)|_{oi} \\ &\wedge \\ mfm_k^0|_{match_o}(av_0).a &= mfm_l^0|_{match_o}(av_1).a \end{aligned}$$

Die einzige Gleichung des Systems *EQATTS*

$$mfm_k^1|match_o(bv_0).b = mfm_l^1|match_o(bv_1).b$$

ist offensichtlich für jede dieser Lösungen erfüllt. Dementsprechend sind die Objektvariablen ov_i und ov_j der Regel r_{appl} konfliktfrei. ❍

Im Folgenden wird eine sehr ähnliche Regel wie im vorangegangenen Beispiel untersucht, die jedoch zwei *nicht* konfliktfreie Objektvariablen enthält:

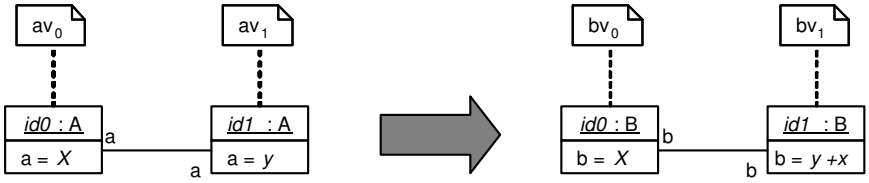


Abbildung 4.9: Regel r_{nappl}

Beispiel: Abbildung 4.9 zeigt die Regel r_{nappl}. Der einzige Unterschied zum vorangegangenen Beispiel ist, dass nun der Attributterm von bv_1 den Wert $x+y$ anstelle von y hat. Daher können die meisten Überlegungen des vorhergehenden Beispiels hier ausgelassen werden und es wird direkt das Gleichungssystem

$$PCONF(\{r_{nappl}\}, bv_0, bv_1, mfm_k^0, mfm_l^0) \wedge VALIDOB(\{r_{nappl}\}, r_{appl}|_{mv_0}|_{mm}, mfm_k^0, mfm_l^0)$$

angegeben:

$$\begin{aligned}
PCONF: \; & mfm_k^1|_{match_o}(bv_0)|_{oi} = mfm_l^1|_{match_o}(bv_1)|_{oi} \\
& mfm_k^1|_{match_o}(bv_0)|_{oi} = mfm_k^0|_{match_o}(av_0)|_{oi} \\
& mfm_k^1|_{match_o}(bv_0).b = mfm_k^0|_{match_o}(av_0).a \\
& mfm_l^1|_{match_o}(bv_1)|_{oi} = mfm_l^0|_{match_o}(av_1)|_{oi} \\
& mfm_l^1|_{match_o}(bv_1).b = mfm_l^0|_{match_o}(av_0).a + mfm_l^0|_{match_o}(av_1).a \\
& \qquad\qquad \wedge \\
VALIDOB: \; \big(\; & mfm_k^0|_{match_o}(av_0)|_{oi} \neq mfm_l^0|_{match_o}(av_1)|_{oi} \; \vee \\
& mfm_k^0|_{match_o}(av_0).a = mfm_l^0|_{match_o}(av_1).a \qquad \big)
\end{aligned}$$

Für jede Lösung von *PCONF* die *VALIDOB* erfüllt muss demnach gelten:

$$\begin{aligned}
& mfm_k^0|_{match_o}(av_0)|_{oi} = mfm_l^0|_{match_o}(av_1)|_{oi} \; \wedge \\
& mfm_k^0|_{match_o}(av_0).a = mfm_l^0|_{match_o}(av_1).a
\end{aligned} \tag{4.22}$$

Das Gleichungssystem $EQATTS(\{r_{nappl}\}, bv_0, bv_1, mfm_k^0, mfm_l^0)$ besteht gemäß Definition 4.1.13 aus einer einzigen Gleichung:

$$EQATTS: \; mfm_k^1|match_o(bv_0).b = mfm_l^1|match_o(bv_1).b$$

In diesem Fall erfüllen nicht alle Lösungen von $PCONF \wedge VALIDOB$ das System *EQATTS*. So führt *EQATTS* hier zu einem neuen Constraint:

$$\underbrace{mfm_k^0|_{match_o}(av_0).a}_{=:x} = \underbrace{mfm_l^0|_{match_o}(av_0).a}_{=:y} + \underbrace{mfm_l^0|_{match_o}(av_1).a}_{=:z}$$

Somit kann festgehalten werden, dass die Objektvariablen ov_i und ov_j der Regel r_{nappl} nicht konfliktfrei sind.

Betrachtet man den neuen Constraint genauer, so lässt sich zunächst feststellen, dass gemäß (4.22) gilt:

$$\underbrace{mfm_k^0|_{match_o}(av_0).a}_{x} = \underbrace{mfm_l^0|_{match_o}(av_1).a}_{z}$$

Hieraus ergibt sich:

$$\underbrace{mfm_l^0|_{match_o}(av_0).a}_{y} = 0$$

Die Regel r_{nappl} ist also nur dann anwendbar, wenn sichergestellt werden kann, dass $mfm_l^0|_{match_o}(av_0).a = 0$ gilt, d.h. jedes von av_0 im Quellmodell gematchte Objekt hat den Wert 0 im Attribut a stehen.

Dies entspricht der Intuition, denn ein Blick auf die Regel zeigt bereits, dass keine Konflikte zwischen Attributwerten auftreten können, wenn av_0 immer auf ein Objekte mit einem Attributwert $a = 0$ matchen würde. Diese Annahme würde jedoch die Menge der möglichen Lösungen weiter einschränken und somit kann die Konfliktfreiheit der beiden Zielobjektvariablen von r_{nappl} nicht nachgewiesen werden. Umgekehrt ließe sich so auch der Nachweis erbringen, dass die Regel sicher nicht anwendbar ist, da bereits jedes Modell mit einem passenden Fragment und $a \neq 0$ für ein Objekt vom Typ A ein Gegenbeispiel darstellt. ❍

4.1.4 Anwendbarkeit von Regeln und Regelwerken

Dieser Abschnitt stellt die Verifikationstechniken zum Nachweis der Anwendbarkeit einer einzelnen Regel und eines Regelwerks vor. Generell ist ein Regelwerk aus anwendbaren Regeln nicht automatisch anwendbar, da immer noch Konflikte zwischen Objekten die von verschiedenen Regeln erzeugt wurden auftreten können.

Satz 4.1.6 (Anwendbarkeit einer Regel)
Eine Regel $r \in \mathbb{R}$ ist anwendbar, falls alle der folgenden drei Kriterien erfüllt sind:

(i) Die Regel erzeugt nur gültige Modellfragmente:

$$createsValidFragments(r)$$

(ii) Alle Zielobjektvariablen sind deterministisch:

$$\forall ov_i \in r|_{mv_1}|_{OVB} : deterministic(r, ov_i))$$

(iii) Alle Zielobjektvariablen sind konfliktfrei:

$$\forall ov_i, ov_j \in r|_{mv_1}|_{OVB} \text{ mit } ov_i \neq ov_j : \; conflictFree(\{r\}, ov_i, ov_j)$$

(Beweis s. A.2.1, S. 292) ❑

Der Nachweis, dass eine Regel nur gültige Modellfragmente erzeugt (Forderung (i) des Satzes) kann wahlweise mit der einfachen Verifikationstechnik aus Satz 4.1.2 oder aber der Verifikationstechnik aus Satz 4.1.1 erbracht werden. Die Gültigkeit von Forderung (ii), welche besagt, dass sämtliche Zielobjektvariablen deterministisch sein müssen, lässt sich mit Hilfe von Satz 4.1.4 belegen. Für den für Forderung (iii) benötigten Nachweis der Konfliktfreiheit aller Zielobjektvariablen stehen wahlweise die einfache Verifikationstechnik aus Lemma 4.1.5 oder aber Satz 4.1.5 zur Verfügung.

Die nachstehende Verifikationstechnik für den Nachweis der Anwendbarkeit eines Regelwerks stützt sich auf die in Satz 4.1.7 vorgestellte Technik zum Nachweis der Anwendbarkeit einzelner Regeln.

Satz 4.1.7 (Anwendbarkeit von Regelwerken)
Ein hinreichendes aber nicht notwendiges Kriterium für die Anwendbarkeit eines Regelwerks $R \in \mathbb{RW}$ ist die Erfüllung der folgenden beiden Forderungen:

(i) $\forall r \in R : r$ ist anwendbar und

(ii) $\forall ov_i, ov_j \in R|_{mv_1}|_{OVB} : \; conflictFree(ov_i, ov_j)$ (Beweis s. A.2.1, S. 293) ❑

Gemäß (i) muss jede einzelne Regel des Regelwerks anwendbar sein. Weiterhin muss gemäß (ii) für jedes Paar aus Zielobjektvariablen des Regelwerks sichergestellt sein, dass sie nicht potentiell zu Konflikten in den Attributwerten eines erzeugten Objekts führen. Für den Nachweis der Konfliktfreiheit von Objektvariablen aus unterschiedlichen Regeln kann wieder wahlweise einfache Verifikationstechnik aus Lemma 4.1.5 oder aber Satz 4.1.5 verwendet werden.

Durch die Möglichkeit die Anwendbarkeit eines Regelwerks zu verifizieren, kann nun überprüft werden ob ein Regelwerk eine Modelltransformation im Sinne von Definition 2.2.1 (Seite 39) spezifiziert: Sei *cm* das in Abschnitt 3.1.2 mathematisch spezifizierte Metametamodell für Klassenmodelle.

Da generell gilt $\mathbb{M} = \overline{\mathbb{M}}^2_{cm}$, wird durch jedes anwendbare Regelwerk mit einem Quellmodell eine Modelltransformation $mt : \overline{\mathbb{M}}^2_{cm} \rightarrow \overline{\mathbb{M}}^2_{cm}$ für objektorientierte Modelle gemäß Definition 2.2.1 spezifiziert. Tatsächlich erzeugen anwendbare Regelwerke nicht nur gültige Modelle aus $\overline{\mathbb{M}}^2_{cm}$, sondern bereits Modellfragmente, welche konsistent zu dem jeweils gegebenen Zielmetamodell sind.

4.2 Metamodellkonformität von Regelwerken

Ist eine Modelltransformation anwendbar, so ist sichergestellt, dass sie aus beliebigen konformen Quellmodellen gültige Modellfragmente erzeugt. Diese Modellfragmente sind konsistent bezüglich des Zielmetamodells, d.h. sie enthalten lediglich Objekte und Assoziationen, deren Typ im Zielmetamodell definiert wurde. Allerdings sind die so erzeugten Modelle nicht notwendigerweise konforme *Modelle* bezüglich des Zielmetamodells gemäß Definition 3.1.14 (S. 61). Ein Modellfragment kann in seinem Metamodell festgelegte Assoziationsmultiplizitäten verletzen.

Im Rahmen dieses Abschnitts werden Verifikationstechniken vorgestellt, die es ggf. erlauben, für ein gegebenes Regelwerk nachzuweisen, dass die bei seiner Anwendung erzeugten Modellfragmente Modelle sind, welche konform zu dem gegebenen Zielmetamodell sind. Hierzu gilt es sicherzustellen, dass in aus einem anwendbaren Regelwerk erzeugten Modell keine Assoziationsmultiplizitäten des Zielmetamodells unter- oder überschritten werden.

4.2.1 Grundlagen

Innerhalb dieses Abschnitts werden eine Reihe von grundlegenden Definitionen und Aussagen, die für den Nachweis der Metamodellkonformität benötigt werden, eingeführt.

Zunächst wird der Begriff Metamodellkonformität eines Regelwerks formal definiert:

Definition 4.2.1 (Metamodellkonformes Regelwerk)
Ein Regelwerk $R \in \mathbb{RW}$ heißt *metamodellkonform*, genau dann wenn das im folgenden definierte Prädikat $mmConform(R)$ gilt.

$$mmConform : \mathbb{RW} \rightarrow \mathbb{B} \text{ mit}$$
$$mmConform(R) :\Leftrightarrow \forall M \in \mathbb{M}^0_R : transform(M,R) \in \mathbb{M}^1_R$$ ❍

D.h. ein Regelwerk ist genau dann metamodellkonform, falls seine Anwendung für beliebige Mengen konformer Quellmodelle immer ein gültiges Modell gemäß Definition 3.1.14 als Ergebnis liefert. Das Regelwerk erzeugt also ausschließlich Modellfragmente, die konform zu seinem Zielmetamodell sind.

Das nachfolgende Lemma hält fest, dass es für den Nachweis der Metamodellkonformität eines anwendbaren Regelwerks genügt zu zeigen, dass keine Assoziationsmultiplizitäten des Metamodells verletzt werden:

Lemma 4.2.1 (Nicht metamodellkonforme Zielmodellfragmente)
Sei $R \in \mathbb{RW}$ ein anwendbares Regelwerk und $M \in \mathbb{M}^0_R$. Dann gilt:

$$transform(M,R) \notin \mathbb{M}^1_R \Rightarrow transform(M,R) \text{ verletzt Assoziationsmultiplizitäten von } mm^1_R.$$

(Beweis s. A.2.2, S. 294) ❑

Liefert also die Anwendung eines anwendbaren Regelwerks ein Modellfragment, welches *kein* gültiges Model bezüglich des Zielmetamodells ist, so verletzt das Modellfragment Assoziationsmultiplizitäten des Zielmetamodells (siehe Definition 3.1.14 von Modellen, Aussage (3.15)).

Im Folgenden werden die beiden Relationen *ubConform* und $lbConform$ eingeführt, die genau dann gelten, wenn ein Modellfragment keine Multiplizitätsober- bzw. -untergrenzen eines Metamodells verletzt. Die in den nachfolgenden Abschnitten vorgestellten Verifikationstechniken zum Nachweis der Ober- bzw. Untergrenzenkonformität von Regelwerken stützen sich auf diese Definitionen, indem sie es erlauben, für ein Regelwerk zu zeigen, dass dieses lediglich obergrenzen- bzw. untergrenzenkonforme Modellfragmente erzeugt.

Die Relation *ubConform* gilt, falls keine Assoziation eines Modellfragments Multiplizitätsobergrenzen eines Metamodells verletzt:

Definition 4.2.2 ($ubConform(mf,mm)$**)**
Für ein Metamodell $mm \in \mathbb{MM}$ und ein hierzu konsistentes Modellfragment $mf \in \mathbb{MF}_{mm}$ ist $ubConform(mf,mm)$ eine Relation, so dass gilt:

$$
\begin{aligned}
&ubConform : \mathbb{MF} \times \mathbb{MM} \to \mathbb{B} \\
&ubConform(mf,mm) :\Leftrightarrow mf \in \mathbb{MF}_{mm} \wedge \\
&\qquad \forall o \in mf|_{OB}, AE \in mm|_{CA}, ae \in AE \text{ mit } ae|_c \in types(o|_{ot}) : \\
&\qquad \Big((lb,ub) := oppositeEnd(AE,ae)|_m \\
&\qquad\qquad \sum_{\substack{oa \in mf|_{OA}: \\ oa|_{OAT}=AE \,\wedge \\ (ae,o)\in oa|_{OAE}}} oa|_{card} \leq ub \qquad \Big)
\end{aligned}
$$

Das Modellfragment wird dann als *obergrenzenkonform* bezeichnet. ❍

$ubConform(mf,mm)$ gilt also genau dann, wenn die Summe aller Kardinalitäten ausgehender Objektassoziationen gleichen Typs für jedes Objekt des Modellfragments mf kleiner oder gleich der Obergrenze aus der entsprechenden Assoziationsmultiplizität aus dem Metamodell mm ist. Da zusätzlich $mf \in \mathbb{MF}_{mm}$ gelten muss, muss das Modellfragment mf konsistent zu dem Metamodell mm sein und darf somit auch nicht den Wert $\perp$ haben.

Analog zu *ubConform* gilt die Relation *lbConform*, falls für keine Klassenassoziation eines Metamodells Multiplizitätsuntergrenzen in einem Modellfragment verletzt werden:

Definition 4.2.3 ($lbConform(mf,mm)$**)**
Für ein Metamodell $mm \in \mathbb{MM}$ und ein hierzu konsistentes Modellfragment $mf \in \mathbb{MF}_{mm}$ ist $lbConform(mf,mm)$ eine Relation, so dass gilt:

$$
\begin{aligned}
&lbConform : \mathbb{MF} \times \mathbb{MM} \to \mathbb{B} \\
&lbConform(mf,mm) :\Leftrightarrow mf \in \mathbb{MF}_{mm} \wedge \\
&\qquad \forall o \in mf|_{OB}, AE \in mm|_{CA}, ae \in AE \text{ mit } ae|_c \in types(o|_{ot}) : \\
&\qquad \Big((lb,ub) := oppositeEnd(AE,ae)|_m \\
&\qquad\qquad lb \leq \sum_{\substack{oa \in mf|_{OA}: \\ oa|_{OAT}=AE \,\wedge \\ (ae,o)\in oa|_{OAE}}} oa|_{card} \qquad \Big)
\end{aligned}
$$

Das Modellfragment wird dann als *untergrenzenkonform* bezeichnet. ❍

Dementsprechend gilt $lbConform(mf, mm)$ genau dann, wenn die Summe aller Kardinalitäten ausgehender Objektassoziationen gleichen Typs für jedes Objekt des Modellfragments *mf* größer oder gleich der Untergrenze aus der entsprechenden Assoziationsmultiplizität aus dem Metamodell *mm* ist.

Um sämtliche Modellfragmente zu erhalten, die während der Anwendung einer Regel r bei einem Quellmodell m_0 erzeugt werden, wird die Menge dieser Fragmente nun als $MFM^1(m_0, r)$ eingeführt.

Definition 4.2.4 ($MFM^1(m,r)$)
Sei $r \in \mathbb{R}$ eine Regel und $m_0 \in \mathbb{M}_{r|_{mv_0}|_{mm}}$ ein mögliches Quellmodell der Regel r.
Dann bezeichnet $MFM^1(m_0, r) \in \mathscr{P}(\mathbb{MFM})$ die Menge von Modellfragment-Matches mfm^1_μ, die während der Anwendung von $apply(m_0, r)$, gemäß Definition 3.5.4 auftreten. ❍

Aufbauend auf der Definition MFM^1 liefert die Funktion *numAssos* die Zahl der durch eine Regelanwendung erzeugten Assoziationen eines gegebenen Typs an einem generierten Objekt.

Definition 4.2.5 ($numAssos(r, m_0, o, ova, ae_1)$)
Es sei $r \in \mathbb{R}$ eine Regel und $m_0 \in \mathbb{M}_{r|_{mv_0}|_{mm}}$ ein mögliches Quellmodell der Regel r. Weiterhin sei $ova \in r|_{mv_1}|_{OVA}$ eine Objektvariablenassoziation der Zielmodellvariablen und $ae_1 \in ova|_{OVAT}$ eines ihrer Enden. Für das Objekt o gelte $o \in apply(m_0, r)$. $MFM^1(m_0, r)$ bezeichne die Menge von Modellfragment-Matches gemäß Definition 4.2.4 bezeichnet. Dann ist die Funktion *numAssos* wie folgt definiert:

$$
\begin{aligned}
&numAssos : \mathbb{R} \times \mathbb{M} \times \mathbb{O} \times \mathbb{OVA} \times \mathbb{OVA}|_{OVAE}|_{ae} \to \mathbb{N}^+ \\
&numAssos(r, m_0, o, ova, ae_1) := \\
&\quad ova|_{cardv} \cdot \Big|\big\{ oa \in apply(m_0, r) : \exists mfm^1_\mu \in MFM^1(m_0, r) \text{ mit } mfm^1_\mu|_{match_a}(ova) = oa \wedge \\
&\qquad\qquad (oppositeEnd(ova|_{OVAT}, ae_1), o) \in oa|_{OAE} \wedge \\
&\qquad\qquad ae_1 \in ova|_{OVAT} \quad \big\}\Big|
\end{aligned}
$$

❍

Für ein Objekt o in einem durch die Regel r aus dem Modell m_0 erzeugten Modellfragment und eine Objektvariablenassoziation *ova* auf der rechten Seite einer Regel liefert *numAssos* die Anzahl aller ausgehenden Assoziationen von o, die direkt von *ova* erzeugt wurden. Das bedeutet für die entsprechenden Assoziationen und die Regel, dass ein Modellfragment-Match mfm^1_μ existiert der eine bijektive Abbildung $mfm^1_\mu|_{match_a}$ beinhaltet. Diese bildet *ova* bijektiv auf *oa* ab. Abbildung 4.10 zeigt ein Beispiel mit einer Objektvariablenassoziation und zwei Objektassoziationen die aus ihr generiert wurden.

4.2.2 Verifikationstechniken für Obergrenzenkonformität

Im Verlauf dieses Abschnitts werden Techniken vorgestellt, die es erlauben, zu verifizieren, dass ein Regelwerk bei gültigen Quellmodellen ausschließlich obergrenzenkonforme Modellfragmente gemäß Definition 4.2.2 (S. 132) erzeugt. Zunächst wird der Begriff *obergrenzenkonformes Regelwerk* formal definiert:

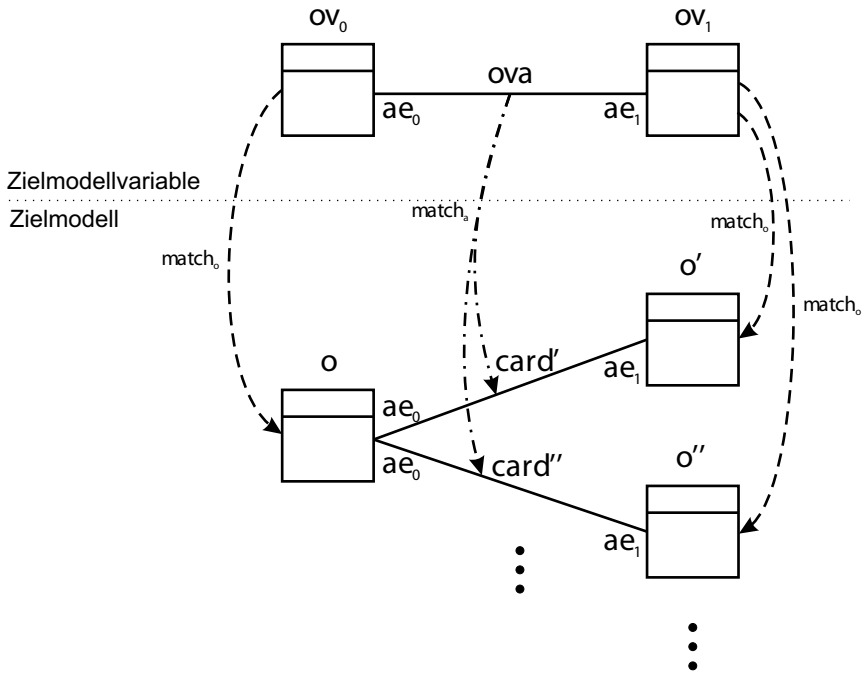


Abbildung 4.10: Eine Objektvariablenassoziationen und die durch sie erzeugten Objektassoziationen

Definition 4.2.6 (Obergrenzenkonformes Regelwerk (*ubConform*(R)))
Ein Regelwerk $R \in \mathbb{RW}$ heißt genau dann *obergrenzenkonform*, wenn das nachstehend definierte Prädikat *ubConform* gilt:

$$ubConform : \mathbb{RW} \rightarrow \mathbb{B}$$
$$ubConform(R) :\Leftrightarrow \forall M \in \mathbb{M}_R^0 : ubConform(transform(M,R), mm_R^1) \qquad \text{❍}$$

Ein Regelwerk ist also genau dann obergrenzenkonform, falls es für beliebige, gültige Quellmodelle ausschließlich obergrenzenkonforme Modellfragmente erzeugt.

Für die Verifikationstechnik zum Nachweis der Obergrenzenkonformität werden eine Reihe von Eigenschaften von Objektvariablen und Objektvariablenassoziationen benötigt. Im einzelnen gilt es die folgenden Fragestellungen zu klären:

- Sind Objektvariablenassoziationen aus verschiedenen Regeln redundant? D.h. kann sichergestellt werden, dass eine Objektvariablenassoziation niemals eine Assoziation zwischen zwei Objekten im Zielmodell erzeugt, die nicht auch von einer anderen Objektvariablenassoziation erzeugt wird, so dass beide Assoziation durch den $\cup_m$-Operator immer zu einer Assoziation im Zielmodell zusammengeführt werden?

- Welche Objektvariablen können mit Sicherheit niemals dasselbe Objekt erzeugen? Für solche Objektvariablen steht fest, dass die von ihnen ausgehenden Objektvariablenassoziationen niemals Assoziationen erzeugen können, die vom selben Objekt im Zielmodell ausgehen.

- Wie oft kann eine Quellmodellvariable in einem beliebigem Quellmodell höchstens matchen, so dass eine Zielobjektvariable derselben Regel immer dasselbe Objekt erzeugt? Mit Hilfe dieser Information ist es möglich festzustellen, wieviele ausgehende Assoziationen gleichen Typs von der jeweiligen Regeln an einem Zielobjekt maximal erzeugt werden können.

In den nachstehenden Abschnitten werden Techniken eingeführt, die es ermöglichen diese Fragestellungen zu beantworten.

Redundante Objektvariablenassoziationen

Für den Nachweis der Obergrenzenkonformität von Regelwerken ist es hilfreich, zunächst Objektvariablenassoziationen in Regeln zu identifizieren, die sich nicht auf die Zahl der erzeugten Assoziationen in einem Zielmodell auswirken. Die Relation *maxRedundant* dient dazu, solche redundanten Assoziationen zu identifizieren:

Definition 4.2.7 (*maxRedundant*(R,r,ova))
Sei $R \in \mathbb{RW}$ ein Regelwerk mit $r \in R$ und $ova \in r|_{mv_1}|_{OVA}$. *maxRedundant*(R,r,ova) ist eine Relation, die in folgender Weise definiert ist:

$$
\begin{aligned}
&\mathit{maxRedundant} : \mathbb{RW} \times \mathbb{R} \times \mathbb{OVA} \\
&\mathit{maxRedundant}(R,r,ova) :\Leftrightarrow \\
&\quad \forall M \in \mathbb{M}_R^0 : \big\{ oa : \exists \mathit{mfm}_\mu^1 \in \mathit{MFM}^1(\mathit{srcModel}(M,r),r) : \\
&\qquad\qquad (\mathit{mfm}_\mu^1|_{\mathit{match_a}}(ova) = oa \wedge oa \in \mathit{transform}(M,R)|_{OA}) \quad \big\} \\
&\qquad\qquad\quad \subseteq \\
&\qquad\qquad \big\{ oa : oa|_{OAT} = ova|_{OVAT} \wedge oa \in \mathit{transform}(m, R \setminus \{r\})|_{OA} \big\}
\end{aligned}
$$

❍

maxRedundant(R,r,ova) gilt demnach genau dann, falls die Objektvariablenassoziation *ova* in der Regel *r* keine Objektassoziationen in beliebigen Zielmodellen erzeugt, die nicht auch von mindestens einer anderen Regeln aus *R* erzeugt wird.

Somit gibt die Relation *maxRedundant* an, ob eine Objektvariablenassoziation Einfluss auf die Zahl der im Zielfragment erzeugten Assoziationen hat. Allerdings ist hierbei zu berücksichtigen, dass aus einer Menge von redundanten Objektvariablenassoziationen eine als relevant anzusehen ist, während die restlichen redundanten Objektvariablenassoziationen vernachlässigbar sind.

Die Relation *maxRelevant* entscheidet ob eine ausgehende Objektvariablenassoziation relevant im Kontext eines gegebenen Regelwerks *R* ist. Dies gilt genau dann, wenn die Objektvariablenassoziation beim generieren neuer Modellfragmente während der Transformation ignoriert werden kann.

Definition 4.2.8 (*maxRelevant*(R,r_i,ova))
Sei $R \in \mathbb{RW}$ ein Regelwerk auf dem eine beliebige aber feste lineare Ordnungsrelation $<$ über den Regel definiert ist. Dann ist die Relation *maxRelevant* folgendermaßen definiert:

$$
\begin{aligned}
&\mathit{maxRelevant} : \mathbb{RW} \times \mathbb{R} \times \mathbb{OVA} \to \mathbb{B} \\
&\mathit{maxRelevant}(R,r_i,ova) \\
&\quad :\Leftrightarrow \quad r_i \in R \ \wedge \ ova \in r_i|_{mv_1}|_{OVA} \wedge \\
&\qquad \neg \Big(\mathit{maxRedundant}(R,r_i,ova) \wedge \\
&\qquad\quad \nexists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge \mathit{maxRedundant}(\{r_j,r_i\},r_i,ova) \big) \Big)
\end{aligned}
$$

❍

Die Objektvariablenassoziation wird nicht berücksichtigt, falls für sie *maxRedundant* gilt und es keine Regel r_j mit $r_j < r_i$ gibt, die eine zur linken Modellvariable r_i strukturell isomorphe Modellvariable hat und zu der r_i *maxRedundant* ist. In diesem speziellen Fall sind die Objektvariablenassoziationen

in den beiden Regeln wechselseitig redundant. Es wird nun die Vereinbarung getroffen, dass lediglich die kleinere Objektvariablenassoziation bezüglich der Relation $<$ berücksichtigt wird.

Da die Relationen *maxRedundant* und *maxRelevant* nicht direkt berechenbar sind, werden im weiteren Verlauf Verfahren vorgestellt, mit denen für einzelne Objektvariablenassoziationen nachgewiesen werden kann, dass sie sicher redundant bzw. nicht relevant sind.

Für diese Verfahren ist es zunächst notwendig, Paare von Quellmodellvariablen in Regelwerken zu identifizieren, bei denen eine Quellmodellvariable eine Substruktur einer anderen Quellmodellvariable bildet. In der Praxis sind solche Fälle nicht unüblich: So wird mit einer Substruktur oftmals nach einem Grundmuster im Quellmodell gesucht, während die Superstruktur weitere Spezialfälle dieses Musters behandelt. Die folgende Definition spezifiziert formal, wann eine Modellvariable eine Substruktur einer anderen Modellvariablen bildet:

Definition 4.2.9 (Substrukturen und isomorphe Modellvariablen ($mv_{sub} \sqsubseteq mv$))
Für zwei gegebene Modellvariablen $mv, mv_{sub} \in \mathbb{MV}$ heißt mv_{sub} eine *Substruktur* von mv, falls die im Folgenden definierte Relation $mv_{sub} \sqsubseteq mv$ gilt:

$$
\begin{aligned}
&\sqsubseteq : \mathbb{MV} \times \mathbb{MV} \to \mathbb{B} \\
&mv_{sub} \sqsubseteq mv :\Leftrightarrow \quad mv_{sub}|_{mm} = mv|_{mm} \\
&\qquad \wedge\ \exists corresponds : mv_{sub}|_{OVB} \to mv|_{OVB} \text{ mit } corresponds \text{ ist injektiv} \\
&\qquad \wedge\ \forall ova_i \in mv_{sub}|_{OVA} : \\
&\qquad\quad \exists_1 ova_j \in mv|_{OVA} : \\
&\qquad\qquad ova_j|_{OVAT} = ova_i|_{OVAT} \ \wedge\ ova_j|_{cardv} = ova_i|_{cardv} \wedge \\
&\qquad\qquad ova_j|_{OVAE}|_{ae} = ova_i|_{OVAE}|_{ae} \wedge \\
&\qquad\qquad \forall ovae' \in ova_j|_{OVAE}, ovae'' \in ova_i|_{OVAE} : \\
&\qquad\qquad\quad ovae'|_{ae} = ovae''|_{ae} \Rightarrow ovae'|_{ov} = corresponds(ovae''|_{ov})
\end{aligned}
$$

Zwei Modellvariablen mv_a und mv_b sind *strukturell isomorph*, genau dann wenn gilt:

$$mv_a \cong mv_b :\Leftrightarrow mv_a \sqsubseteq mv_b \wedge mv_b \sqsubseteq mv_a$$

Es seien mv_a und mv_b Modellvariablen. Dann gilt weiter:

$$mv_a \sqsubset mv_b :\Leftrightarrow mv_a \sqsubseteq mv_b \wedge \neg(mv_a \cong mv_b)$$ ❍

Also gilt $mv_{sub} \sqsubseteq mv$ dann, wenn mv_{sub} strukturell ein Teil der Modellvariable mv ist. Die Terme in den Attributen und Identifikatoren können sich hierbei jedoch unterscheiden. Die linke Modellvariable von r_1 in Abbildung 4.11 (siehe Seite 138) ist eine Substruktur der linken Modellvariable von $r0$, d.h. es gilt $r_1|_{mv_0} \sqsubseteq r_0|_{mv_0}$.

Dementsprechend weiß man, dass jeder Match der Superstruktur auch einen Match der Substruktur impliziert. Diese Eigenschaft wird ausgenutzt, um redundante Objektvariablenassoziationen identifizieren zu können. Das folgende Lemma hält diese Eigenschaft von Substrukturen fest:

Lemma 4.2.2 (Matches von Substrukturen in Quellmodellen)
Sei $R \in \mathbb{RW}$ ein Regelwerk, $r_i, r_j \in R$ mit $r_i|_{mv_0} \sqsubseteq r_j|_{mv_0}$ und *corresponds* eine injektive Abbildung gemäß Definition 4.2.9. Weiter sei $mm := r_i|_{mv_0}|_{mm} = r_j|_{mv_0}|_{mm}$ das Quellmetamodell der beiden Regeln. Dann gilt:

$$
\begin{aligned}
&\forall m \in \mathbb{M}_{mm}, mfm^j \in MFM(m, r_j|_{mv_0}) : \\
&\quad \exists mfm^i \in MFM(m, r_i|_{mv_0}) :\ mfm^i|_{match_o} \circ corresponds = mfm^j|_{match_o}
\end{aligned}
$$

(Beweis s. A.2.2, S. 294) ❑

Es bleibt festzuhalten, dass die Werte von Identifikator- und Attributvariablen hierbei nicht berücksichtigt werden. Demzufolge gelten beispielsweise auch Modellvariablen als isomorph, bei denen eine Constraint-behaftet ist und die andere nicht. Formal gesehen ist dies korrekt, da die Modellfragment-Match-Menge auch Matches beinhaltet, bei denen Constraints der Quellmodellvariablen verletzt werden. Gemäß Definition 3.5.5 (S. 89) liefern diese jedoch, im Gegensatz zu Matches ohne verletzten Constraint, immer den Wert $(\emptyset,\emptyset)$ als Ergebnis einer Modellfragmenttransformation. Wie sich jedoch im weiteren Verlauf dieses Abschnitts zeigen wird, hat dieser Umstand keinen Einfluss auf die Korrektheit der Verifikationstechniken, die auf Lemma 4.2.2 basieren.

Die Eigenschaft aus Definition 4.2.9 wird nun verwendet, um für zwei Regeln, deren eine Quellmodellvariable eine Substruktur der zweiten ist, festzulegen, welche Objektvariablen der jeweils rechten Regelseite dasselbe Objekt erzeugen, falls die Substruktur auf dieselben Objekte wie die Superstruktur matcht und keine Constraints verletzt wurden.

Definition 4.2.10 ($OVP(r_{super}, r_{sub})$)
Seien $r_{super}, r_{sub} \in R$ zwei Regeln eines Regelwerks $R \in \mathbb{RW}$ für die gilt:

$$r_{sub}|_{mv_0} \sqsubseteq r_{super}|_{mv_0}$$

Gemäß Definition 4.2.9 muss eine nicht-leere Menge von injektiven *corresponds*-Abbildungen existieren, die im Folgenden mit $\{corresponds_0, \ldots, corresponds_m\}$ bezeichnet wird.
Sei

$$OVP_k(r_{super}, r_{sub}) := \{(ov^0_{super}, ov^0_{sub}), \ldots, (ov^n_{super}, ov^n_{sub})\}$$

(mit $k \in \{0,\ldots,m\}$) die Menge von Paaren von Objektvariablen für die gilt:

$$(i) \quad \forall i \in \{0,\ldots,n\}: \quad ov^i_{sub} \in r_{sub}|_{mv_1|_{OVB}} \quad \wedge \quad ov^i_{super} \in r_{super}|_{mv_1|_{OVB}} \wedge \quad ov^i_{super}|_{otv} = ov^i_{sub}|_{otv}$$

(ii) Für die beiden Regeln r_{super} und r_{sub} hat das Gleichungssystem *GL* aus Definition 3.5.4 dieselbe Lösung für alle Objektidentifikatoren von Objektvariablen die in OVP_k vorkommen, d.h.

$$\forall i \in \{0,\ldots,n\}: \quad match^1_o(ov^i_{super})|_{oi} = match^1_o(ov^i_{sub})|_{oi}[ov'/corresponds_k(ov')]_{ov' \in r_{sub}|_{mv_0}|_{OVB}} \tag{4.23}$$

$t(a)[a/b]$ bezeichnet hierbei die Termsubstitution, welche jedes a in $t(a)$ durch b ersetzt.
Dann bezeichnet

$$OVP(r_{super}, r_{sub}) := \{OVP_0(r_{super}, r_{sub}), \ldots, OVP_m(r_{super}, r_{sub})\}$$

die Menge aller möglichen $OVP_k(r_{super}, r_{sub})$.

Falls $r_{sub}|_{mv_0} \sqsubseteq r_{super}|_{mv_0}$ nicht gilt, liefert $OVP(r_{super}, r_{sub})$ das Ergebnis $\emptyset$. ❍

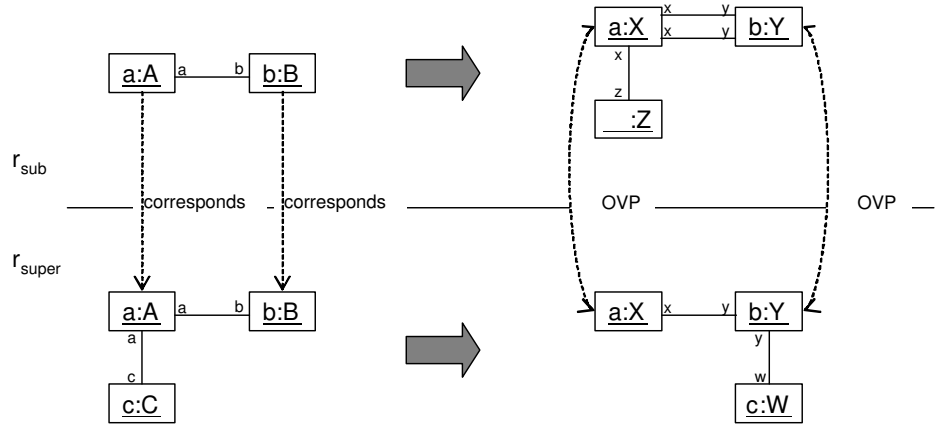


Abbildung 4.11: Ein Beispiel für $OVP(r_{super}, r_{sub})$

$OVP(r_{super}, r_{sub})$ liefert Mengen von Tupeln aus Zielobjektvariablen, die immer dieselben Zielobjekte erzeugen, falls die Quellmodellvariablen auf die gleichen Objekte (ausgenommen die nicht in der Substruktur enthalten) matchen.

Verfügen die Typen dieser Zielobjektvariablen über Primärschlüsselattribute, so ergibt sich in (4.23) für den Wert von $match^1_{\mathrm{o}}(ov^i_{super})|_{id}$ bzw. $match^1_{\mathrm{o}}(ov^i_{sub})|_{id}$ jeweils ein Term der Form

$$pk((att_0, term_0), (att_1, term_1), \ldots (att_n, term_n))$$

(siehe Definition 3.1.11 (3.11)). Auch hier werden die in den jeweiligen Termen von r_{sub} vorkommenden Objektvariablen durch solche aus r_{super} substituiert. Somit ist sichergestellt, dass die Paare aus Zielobjektvariablen jeweils identische Werte in ihren Primärschlüsselattributen haben müssen.

Es bleibt anzumerken, dass die Möglichkeit von Konflikten zwischen Attributwerten dieser neu erzeugte Objekte hier nicht berücksichtigt wird, da diese Eigenschaft für ein Regelwerk bereits durch Verifikationstechniken zur Sicherung der Anwendbarkeit (siehe Abschnitt 4.1) überprüft worden sein sollte.

Abbildung 4.11 zeigt ein Beispiel, in dem $r_{sub}|_{mv_0}$ eine Substruktur von $r_{super}|_{mv_0}$ darstellt. Eine gültige *corresponds* Abbildung zwischen den beiden linken Modellvariablen wird durch die gestrichelten Pfeile angedeutet. Zwischen den beiden rechten Modellvariablen sind die Tupel aus *OVP* durch gestrichelte Doppelpfeile markiert.

OVP wird nun genutzt, um eine Technik angeben zu können, mit der sich Objektvariablenassoziationen identifizieren lassen, für die *maxRedundant* sicher gilt:

Definition 4.2.11 ($redundant(R, r_i, ova)$**)**
Es sei $R \in \mathbb{RW}$ ein Regelwerk mit $r_i \in R$ und $ova_i \in r_i|_{mv_1}|_{OVA}$.
$redundant(R, r_i, ova_i) :\Leftrightarrow$

(i) $\exists\, r_j \in R : OVP(r_i, r_j) \neq \emptyset$, (dies impliziert bereits, dass gilt: $r_j|_{mv_0} \sqsubseteq r_i|_{mv_0}$) $\wedge$

(ii) $\exists\, cardv_j \in \mathbb{N}^+, ova_j \in r_j|_{mv_1}|_{OVA}, OVP_k(r_i, r_j) \in OVP(r_i, r_j)$:

$$cardv_j \geq ova_i|_{cardv} \wedge$$
$$ova_j = (ova_i|_{OVAT}, cardv_j, OVAE_j) \text{ mit}$$
$$OVAE_j = \{(ae, ov_j) : \exists (ae, ov_i) \in ova_i|_{OVAE} \wedge (ov_i, ov_j) \in OVP_k(r_i, r_j)\}$$ ❍

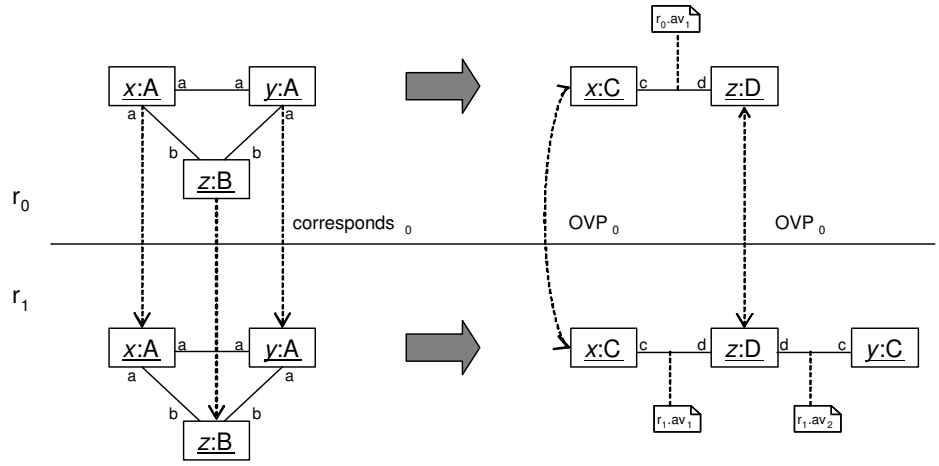


Abbildung 4.12: Eines von zwei möglichen Mappings zwischen r_0 und r_1 und die entsprechende Menge OVP_0.

Für eine gegebenes Regelwerk R und eine Regel r_i, die eine Objektvariablenassoziation *ova* auf der rechten Seite enthält, liefert *redundant* den Wert *true* zurück, falls *ova* mit Sicherheit redundant ist. D.h. alle Instanzen dieser Assoziation würden ohnehin auch von mindestens einer anderen Regel erzeugt werden. Dementsprechend muss *ova* nicht mehr weiter berücksichtigt werden, wenn die maximal möglichen ausgehenden Assoziationen in generierten Modellen abgeschätzt werden.

Im folgenden Lemma wird gezeigt, dass *redundant* eine gültige Abschätzung für *maxRedundant* ist, d.h. für alle Objektvariablenassoziationen für die *redundant* gilt, gilt auch die „ideale“ Eigenschaft *maxRedundant*.

Lemma 4.2.3 (*redundant* impliziert *maxRedundant*)
Es sei $R \in \mathbb{RW}$ ein anwendbares Regelwerk mit $r \in R$ und $ova \in r|_{mv_1}|_{OVA}$. Dann gilt:

$$redundant(R,r,ova) \Rightarrow maxRedundant(R,r,ova) \qquad \text{(Beweis s. A.2.2, S. 294)} \ \square$$

Beispiel: In diesem Beispiel werden zwei Regeln r_0 und r_1 untersucht, für die der Sonderfall $r_0|_{mv_0} \cong r_1|_{mv_0}$ gilt. Die beiden Regeln sind in Abbildung 4.12 dargestellt. Es lässt sich leicht zeigen, dass $r_0|_{mv_0} \sqsubseteq r_1|_{mv_0}$ gemäß Definition Definition 4.2.9 gilt, indem die injektive Abbildung $corresponds_0$, wie auf der linken Seite der Abbildung dargestellt, angegeben wird. Darüber hinaus lässt sich jedoch noch eine zweite mögliche Abbildung $corresponds_1$ angeben, die auf der linken Seite von Abbildung 4.13 dargestellt ist.

Daher enthält $OVP(r_1,r_0) = \{OVP_0(r_1,r_0), OVP_1(r_1,r_0)\}$ *zwei* Mengen von Tupeln, eine für jede der beiden möglichen *corresponds*-Abbildungen. Die erste Menge OVP_0 ist in Form von Doppelpfeilen auf der rechten Seite von Abbildung 4.12 dargestellt, die zweite mögliche Tupelmenge OVP_1 ist auf der rechten Seite von Abbildung 4.13 zu sehen.

Demnach lässt sich *redundant* für die Objektvariablenassoziationen auf der rechten Seite von r_1 für das Regelwerk $R = \{r_0, r_1\}$ berechnen. Definition 4.2.11 (i) gilt offensichtlich, da bekannt ist, dass gilt $OVP(r_1,r_0) \neq \emptyset$.

Für $redundant(R,r_1,r_1.av_1)$ kann gezeigt werden, dass die Objektvariablenassoziation $r_0.av_1$ in r_0 die von Definition 4.2.11 (ii) geforderten Eigenschaften erfüllt, wenn $OVP_0(r_1,r_0)$ als das $OVP_k(r_1,r_0)$ aus Definition 4.2.11 (ii) angenommen wird.

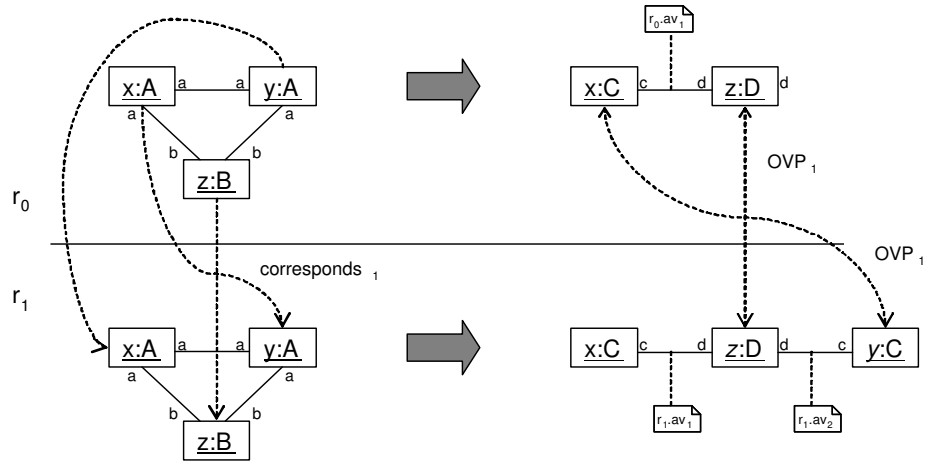


Abbildung 4.13: Das zweite mögliche Mapping zwischen r_0 und r_1 und die entsprechende Menge OVP_1.

Um zu zeigen, dass auch $redundant(R, r_1, r_1.av_2)$ gilt, muss $OVP_0(r_1, r_0)$ als das $OVP_k(r_1, r_0)$ aus Definition 4.2.11 (ii) angenommen werden.

Dementsprechend sind beide Objektvariablenassoziationen aus r_1 redundant, d.h. sie erzeugen nur Assoziationen die ohnehin von einer anderen Regel (in diesem Fall r_0) erzeugt werden würden. Allerdings kann analog auch gezeigt werden, dass die Objektvariablenassoziation $r_0.av_1$ aus Regel r_0 ebenfalls redundant ist. Dies ist der Fall, da r_0 und r_1 genau dieselben Assoziationen im Zielmodell erzeugen. ❍

Auf Basis der in Definition 4.2.11 eingeführten Abschätzung für *maxRedundant* wird nun die Relation *relevant* eingeführt, welche es erlaubt eine Klasse von Objektvariablen zu identifizieren, die mit Sicherheit sämtliche Zielobjektvariablen eines Regelwerks enthält, für die *maxRelevant* gilt.

Definition 4.2.12 (*relevant*(R, r_i, ova)**)**
Sei $R \in \mathbb{RW}$ ein Regelwerk auf dem eine beliebige aber feste lineare Ordnungsrelation $<$ über den Regel definiert ist.

$$
\begin{aligned}
&relevant : \mathbb{RW} \times \mathbb{R} \times \mathbb{OVA} \rightarrow \mathbb{B} \\
&relevant(R, r_i, ova) \ :\Leftrightarrow \\
&\quad r_i \in R \ \wedge ova \in r_i|_{mv_1}|_{OVA} \ \wedge \\
&\quad \neg\Big(redundant(R, r_i, ova) \ \wedge \ \nexists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \ \wedge \ redundant(\{r_j, r_i\}, r_i, ova)\big)\Big) \ ❍
\end{aligned}
$$

relevant erhält eine Objektvariablenassoziation, die Regel welche sie enthält und das Regelwerk als Eingabe. Ähnlich wie *maxRelevant* entscheidet die Relation *relevant*, ob eine Objektvariablenassoziation relevant im Kontext eines gegebenen Regelwerks R ist. Dies ist der Fall, falls die Assoziation nicht ignoriert werden kann, ohne die möglichen Ergebnisse einer Modelltransformation zu ändern. Während *maxRelevant* immer das optimale Ergebnis liefert, verwendet *relevant* die leichter berechenbare Abschätzung *redundant*, die es erlaubt, redundante Objektvariablenassoziationen zu erkennen. Die Eigenschaft, dass *relevant* eine geeignete Abschätzung für *maxRelevant* ist wird im nachstehenden Lemma festgehalten:

Satz 4.2.1 (*maxRelevant* impliziert *relevant*)
Sei $R \in \mathbb{RW}$ ein Regelwerk, $r \in R$ und $ova \in r|_{mv_1}|_{OVA}$. Dann gilt:

$$maxRelevant(R,r,ova) \Rightarrow relevant(R,r,ova)$$ (Beweis s. A.2.2, S. 295) ❑

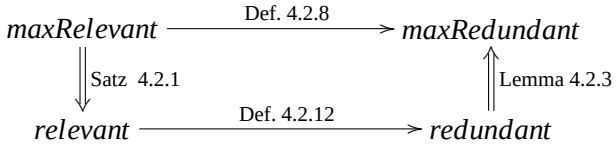


Abbildung 4.14: Abhängigkeiten zwischen *maxRelevant*, *maxRedundant* und ihren (pessimistischen) Schätzheuristiken *relevant* und *redundant*

In Abbildung 4.14 sind die Abhängigkeiten zwischen *relevant*, *redundant*, und ihren „idealen“ Verwandten dargestellt. In der Praxis müssen die Ergebnisse der idealen Relationen *maxRelevant* und *maxRedundant* durch die leicht berechenbaren, pessimistischeren Relationen *relevant* und *redundant* abgeschätzt werden. Wird eine Objektvariablenassoziation als *redundant* erkannt, so ist gemäß Lemma 4.2.3 sichergestellt, dass auch *maxRedundant* für sie gilt. Umgekehrt gilt für alle Objektvariablenassoziationen, welche *maxRelevant* sind, dass sie auch durch die Abschätzung *relevant* als solche erkannt werden.

Identifizierung von Objektvariablen die keine identischen Objekte erzeugt können

Im Folgenden gilt es, innerhalb eines Regelwerks Objektvariablen der rechten Regelseiten zu identifizieren, welche mit Sicherheit niemals Objekte mit derselben Identität erzeugen können. Dementsprechend lassen sich Klassen von Objektvariablen bilden, wobei aus allen Objektvariablen einer solchen Klasse potentiell dasselbe Objekt im Zielmodell entstehen kann. In den nachfolgenden Abschnitten wird untersucht, wieviele Assoziationen maximal von einem erzeugten Objekt ausgehen können. Hierzu wird berechnet, wieviele Assoziationen durch jede Regel an einem, durch eine Zielobjektvariable der Regel erzeugten Objekt, erzeugt werden können. Sind die verschiedenen Klassen von Objektvariablen identifiziert, so können Objektvariablen aus verschiedenen Klassen hierbei getrennt untersucht werden. Dies erlaubt letztendlich eine bessere Abschätzung und erhöht somit die Zahl der Regelwerke, für die sich die Obergrenzenkonformität nachweisen lässt.

Die nachfolgende Definition führt die Eigenschaft $createSameObj$ ein. Diese gilt genau dann für eine Menge von Zielobjektvariablen eines Regelwerks, wenn eine Menge von Quellmodellen existiert, so dass alle Objektvariablen zumindest einmal dasselbe Zielobjekt erzeugen.

Definition 4.2.13 ($createSameObj(R,\{ov_0,\ldots,ov_n\})$)
Sei $R \in \mathbb{RW}$ ein Regelwerk und $\{ov_0,\ldots,ov_n\} \subseteq R|_{mv_1}|_{OVB}$ eine Menge von Zielobjektvariablen des Regelwerks. Dann ist die Relation $createSameObj$ folgendermaßen definiert:

$$createSameObj : \mathbb{RW} \times \mathscr{P}(\mathbb{OV}) \rightarrow \mathbb{B}$$
$$createSameObj(R,\{ov_0,\ldots,ov_n\}) :\Leftrightarrow$$
$$\exists M \in \mathbb{M}_R^0 :$$
$$\forall ov_i \in \{ov_0,\ldots,ov_n\} :$$
$$\exists r \in R \text{ mit } ov_i \in r|_{mv_1}|_{OVB}, o \in transform(M,R)|_{OB} :$$
$$createsObj(o,ov_i,r,srcModel(M,r))$$ ❍

createSameObj erhält eine Menge von Zielobjektvariablen mit den zugehörigen Regeln und gilt genau dann, falls es eine beliebige aber feste Menge gültiger Quellmodelle gibt, für die jede Objektvariable der Menge dasselbe Objekt im Zielmodell erzeugt. Interessant ist es zu wissen, ob eine solche Menge aus Zielobjektvariablen maximal ist. Definition 4.2.14 führt eine dementsprechende Relation für Regelwerke und Mengen von Objektvariablen ein:

Definition 4.2.14 (*maxCreateSameObj*$(R,\{ov_0,\ldots,ov_n\})$)
Sei $R \in \mathbb{RW}$ ein Regelwerk und $\{ov_0,\ldots,ov_n\} \subseteq R|_{mv_1}|_{OVB}$ eine Menge von Zielobjektvariablen des Regelwerks. Dann ist die Relation *maxCreateSameObj* folgendermaßen definiert:

$$
\begin{aligned}
&\mathit{maxCreateSameObj} : \mathbb{RW} \times \mathscr{P}(\mathbb{OV}) \to \mathbb{B} \\
&\mathit{maxCreateSameObj}(R,\{ov_0,\ldots,ov_n\}) :\Leftrightarrow \\
&\quad \mathit{createSameObj}(R,\{ov_0,\ldots,ov_n\}) \wedge \\
&\quad \nexists OV' \in \mathscr{P}(\mathbb{OV}) : \left(\{ov_0,\ldots,ov_n\} \subset OV' \wedge \mathit{createSameObj}(R,OV')\right)
\end{aligned}
$$ ❍

maxCreateSameObj gilt, falls die Zielobjektvariablen eine maximale Menge innerhalb eines Regelwerks darstellen, für die *createSameObj* gilt. D.h. es existiert keine Menge OV' von Objektvariablen, die eine Obermenge von $\{ov_0,\ldots,ov_n\}$ darstellt und für die ebenfalls *createSameObj* gilt.

Definition 4.2.15 definiert eine Heuristik um festzustellen, wann eine Menge von Zielobjektvariablen $\{ov_0,\ldots,ov_n\}$ aus R *sicherlich nicht* alle ein identisches Objekt im Zielmodell anlegen können.

Definition 4.2.15 (*cannotCreateSame*$(R,\{ov_0,\ldots,ov_n\})$)
Sei $R \in \mathbb{RW}$ ein Regelwerk und $\{ov_0,\ldots,ov_n\} \subseteq R|_{mv_1}|_{OVB}$ eine Menge von Zielobjektvariablen des Regelwerks. Dann ist die Relation *cannotCreateSame* folgendermaßen definiert:

$$
\begin{aligned}
&\mathit{cannotCreateSame} : \mathbb{R} \times \mathscr{P}(\mathbb{OV}) \to \mathbb{B} \\
&\mathit{cannotCreateSame}(R,\{ov_0,\ldots,ov_n\}) :\Leftrightarrow
\end{aligned}
$$

(i) $\{ov_0,\ldots,ov_n\} \not\subseteq R|_{mv_1}|_{OVB} \vee$

(ii) $\exists ov_i, ov_j \in \{ov_0,\ldots,ov_n\}$ mit $ov_i \neq ov_j : ov_i \not\sim ov_j \vee$

(iii) $\exists OV^0 \subseteq R|_{mv_0}|_{OVB}, OV^1 \subseteq \{ov_0,\ldots,ov_1\}$:

„Alle Objektvariablen aus OV^1 erzeugen dasselbe Objekt."
$\Rightarrow$
„Alle Objektvariablen aus OV^0 matchen auf dasselbe Quellobjekt."

Sei mm_0 das gemeinsame Quellmetamodell der Objektvariablen aus OV^0 und *class* ihr Typ. Dann gilt:

$$
\begin{aligned}
&\exists AE \in mm_0|_{CA}, ae \in AE \text{ mit } ae|_c \in \mathit{types}(\mathit{class}) : \\
&\quad (lb,ub) := \mathit{oppositeEnd}(AE,ae)|_m \wedge \\
&\quad ub < \sum_{card \in \mathbb{N}^+} \Bigg(\max_{ov^0 \in OV^0} \Big\{ \sum_{\substack{ova_k \in ova_k \in R|_{mv_0}|_{OVA} : \\ ova_k|_{OVAT} = AE \wedge \\ (ae,ov_i^0) \in ova_k|_{OVAE} \wedge \\ ova_k|_{OVAE}|_{ov} \neq \{ov_i^0\} \wedge \\ ova_k|_{cardv} = card}} card \Big\} + \max_{ov^0 \in OV^0} \Big\{ \sum_{\substack{ova_k \in ova_k \in R|_{mv_0}|_{OVA} : \\ ova_k|_{OVAT} = AE \wedge \\ (ae,ov_i^0) \in ova_k|_{OVAE} \wedge \\ ova_k|_{OVAE}|_{ov} = \{ov_i^0\} \wedge \\ ova_k|_{cardv} = card}} card \Big\} \Bigg)
\end{aligned}
$$ ❍

Fall (i) stellt lediglich sicher, dass alle Objektvariablen $\{ov_0,\ldots,ov_n\}$ aus rechten Regelseiten des Regelwerks R stammen.

Fall (ii) legt fest, dass zwei Objektvariablen kein identisches Objekt erzeugen können, falls sie nicht ähnlich gemäß Definition 3.5.7 sind. Dies impliziert auch, dass keiner der Identifikatorterme den Wert $\diamondsuit$ haben kann, falls mehr als eine Objektvariable als Eingabe für *cannotCreateSame* existiert.

In Fall (iii) wird untersucht, ob für die Erzeugung eines identischen Objektes durch Objektvariablen aus $\{ov_0,\ldots,ov_n\}$ jeweils dieselben Quellobjekte gematcht werden müssen. Dieser Fall ist in der Praxis relativ häufig anzutreffen, beispielsweise wenn der Identifikator eines Quellobjektes durch verschiedenen Regeln in ein Zielobjekt kopiert wird.

Ist dies der Fall, so wird überprüft, ob es ein solches Quellobjekt, das von den Objektvariablen aus OV^0 gematcht wird, geben kann. Dies ist genau dann nicht der Fall, wenn die Zahl der ausgehenden Assoziationen eines Typs, die gemäß der verschiedenen Quellmodellvariablen existieren müssten, die Zahl der gemäß dem Quellmetamodell erlaubten überschreitet.

Man weiß, dass Objektvariablenassoziationen mit unterschiedlicher Kardinalität, sowie reflexive und nicht reflexive Objektvariablenassoziationen nicht auf dieselben Assoziationen des Quellmodells matchen können. Übersteigt also die Anzahl, der bei den Quellmodellvariablen gefundenen verschiedenen Objektvariablenassoziationen eines Typs, die maximal mögliche Anzahl *ub* ausgehender Assoziationen, so ist es nicht möglich, dass die betrachteten Objektvariablen der linken Seite alle gleichzeitig auf dasselbe Objekt matchen. Folglich können sie auch nicht alle dasselbe Objekt erzeugen. Im Folgenden wird dieser Sachverhalt anhand eines Beispiels verdeutlicht:

Beispiel: Abbildung 4.15 zeigt ein Regelwerk R aus drei Regeln, für im Folgenden untersucht wird, ob $cannotCreateSame(R,\{r_0.ov0_0,r_1.ov0_1,r_2.ov0_2\})$ gilt. Offensichtlich ist diese Eigenschaft nicht mit Hilfe des ersten der beiden Fälle aus Definition 4.2.15 nachweisbar. (ii) trifft nicht zu, da die Objektvariablen ov_0, ov_1 und ov_2 ähnlich, d.h. es gilt:

$$ov_0 \sim ov_1 \sim ov_2$$

Das Quellmetamodell der drei Regeln ist in Abbildung 4.16 dargestellt. Eine Menge, für welche die in (iii) aufgestellte Implikation offensichtlich gilt, ist die Menge $OV^0 = \{ov0_0, ov0_1, ov0_2\}$.

Es wird nun das Assoziationsende $a0$ der dort abgebildeten, reflexiven Assoziation untersucht. Das Ergebnis für die einzelnen Werte der in der Ungleichung gebildeten Summen ist in Tabelle 4.1 dargestellt.

Da keine Kardinalitäten von Objektvariablenassoziationen größer als 2 in den Regeln vorkommen, ergeben alle weiteren Summen den Wert 0. Folglich errechnet sich die Gesamtsumme der Ungleichung aus Definition 4.2.15 (iii) folgendermaßen:

$$\begin{aligned}
&\max\{2\cdot 1,0\cdot 1,0\cdot 1\}+\max\{0\cdot 1,0\cdot 1,1\cdot 1\}+ && (card=1)\\
&\max\{0\cdot 2,1\cdot 2,0\cdot 2\}+\max\{0\cdot 2,0\cdot 2,0\cdot 2\} && (card=2)\\
=&2+1+2\\
=&5\\
>&2=ub
\end{aligned}$$

Dementsprechend können also nicht alle drei Regeln r_0, r_1 und r_2 dasselbe Objekt mit den Objektvariablen ov_0, ov_1 und ov_2 erzeugen, obwohl sie unabhängig voneinander im Quellmodell matchen könnten.

Der vorliegende Sachverhalt wird anschaulich, wenn man sich überlegt wie ein Modellfragment beschaffen sein müsste auf das die linken Seiten aller Regeln matchen könnten. Es müsste zumindest

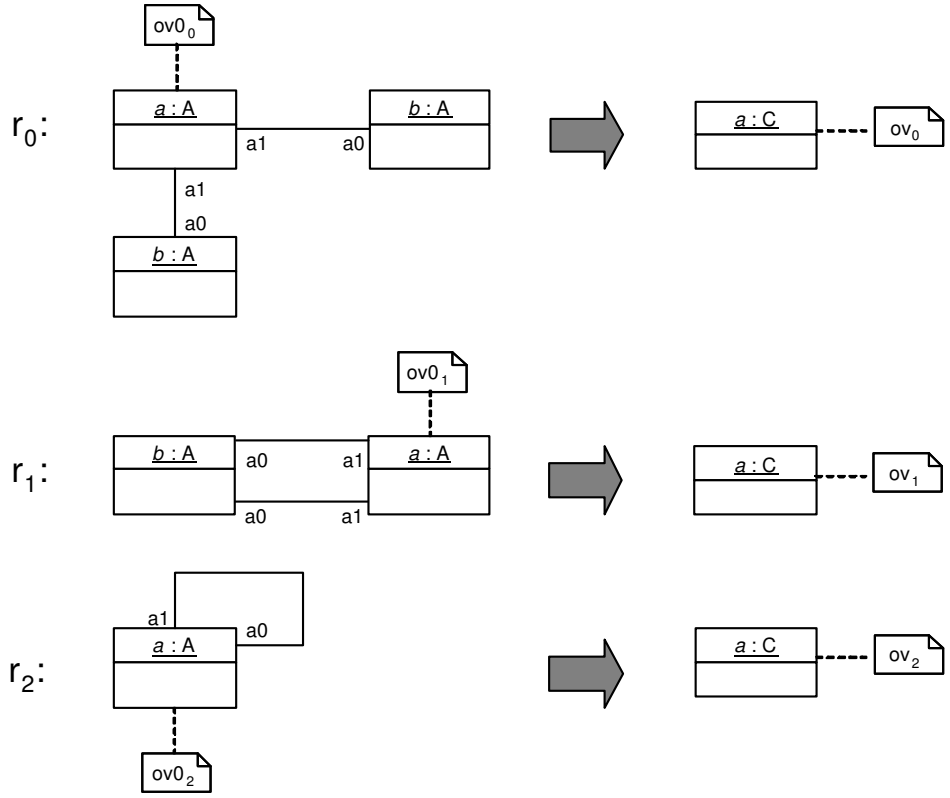


Abbildung 4.15: Drei Modelltransformationsregeln

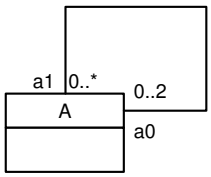


Abbildung 4.16: Das gemeinsame Quellmetamodell der drei Regeln

card	ov^0	1. Summe	2. Summe
1	$ov0_0$	$2 \cdot card$	$0 \cdot card$
1	$ov0_1$	$0 \cdot card$	$0 \cdot card$
1	$ov0_2$	$0 \cdot card$	$1 \cdot card$
2	$ov0_0$	$0 \cdot card$	$0 \cdot card$
2	$ov0_1$	$1 \cdot card$	$0 \cdot card$
2	$ov0_2$	$0 \cdot card$	$0 \cdot card$
3	$ov0_0$	$0 \cdot card$	$0 \cdot card$
3	$ov0_1$	$0 \cdot card$	$0 \cdot card$
...	...	...	...

Tabelle 4.1: Berechnung der Summanden für Fall (iii) von *cannotCreateSame*

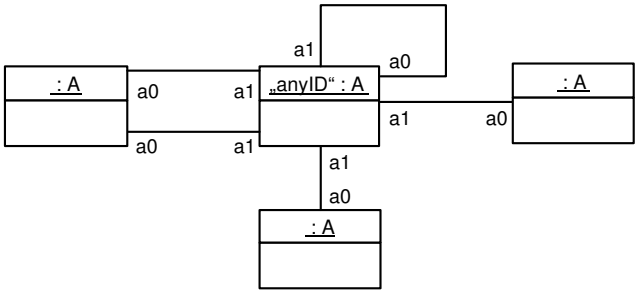


Abbildung 4.17: Ein Ausschnitt eines Quellmodells bei dem ov_0, ov_1 und ov_2 jeweils auf das Objekt mit der Identität *anyID* matchen würden.

die in Abbildung 4.17 dargestellte Struktur aufweisen. Offensichtlich ist dieses Quellmodellfragment aber nicht konform zum Quellmetamodell, da es die zulässige Zahl von maximal 2 ausgehenden $a0$-Assoziationen überschreitet und statt dessen über 5 solche Assoziationen verfügt. ❍

Das nachfolgende Lemma besagt, dass die in Definition 4.2.15 eingeführte Heuristik *cannotCreateSame* eine gültige Abschätzung für $\neg createSameObj$ darstellt.

Lemma 4.2.4 (*cannotCreateSame* impliziert $\neg createSameObj$)
Es gilt für $R \in \mathbb{RW}, \{ov_0,\ldots,ov_n\} \subseteq \mathbb{OV}$:

$$cannotCreateSame(R,\{ov_0,\ldots,ov_n\}) \Rightarrow \neg createSameObj(R,\{ov_0,\ldots,ov_n\})$$

(Beweis s. A.2.2, S. 296) ❑

Definition 4.2.16 ($\mathbb{OV}_R^{conf}$)
Sei $R \in \mathbb{RW}$, $\mathbb{OV}_R := \mathscr{P}(R|_{mv_1}|OVB)$. Dann gelte:

$$\mathbb{OV}_R^{conf} := \{OV \in \mathbb{OV}_R : maxCreateSameObj(R,OV)\}$$ ❍

Die Menge $\mathbb{OV}_R$ enthält alle möglichen Mengen aus Zielobjektvariablen eines Regelwerks R. $\mathbb{OV}_R^{conf}$ enthält alle maximalen Mengen aus Zielobjektvariablen eines Regelwerks R, für die Quellmodelle existieren, so dass mindestens ein Objekt des Zielmodells durch jede der Zielobjektvariablen mindestens einmal erzeugt wird. Diese Menge ist besonders interessant, da sämtliche ausgehenden Assoziationen aus diesem Objekt ebenfalls nur von den gefundenen Regeln erzeugt werden können. Regeln, die ein solches Objekt nicht erzeugen, brauchen demnach, bei der Abschätzung der möglichen maximalen Anzahl ausgehender Assoziationen, lediglich im Kontext eines anderen Elements aus $\mathbb{OV}_R^{conf}$ betrachtet zu werden. Dies erlaubt eine schärfere Abschätzung der möglichen maximalen Anzahl ausgehender Assoziationen als in [BM03a], in dem sämtliche Kardinalitäten von Objektvariablenassoziationen aus allen Regeln aufaddiert werden.

Definition 4.2.17 stellt eine Abschätzung für $\mathbb{OV}_R^{conf}$ vor, die auf der Relation *cannotCreateSame* basiert:

Definition 4.2.17 ($\mathbb{OV}_R^{conf*}$)
Sei $R \in \mathbb{RW}$, $\mathbb{OV}_R := \mathscr{P}(R|_{mv_1}|_{OVB})$. Dann gelte:

$$\mathbb{OV}_R^{conf*} := \mathbb{OV}_R \backslash \{OV \in \mathbb{OV}_R : cannotCreateSame(R,OV)\}$$
$$\text{mit } \forall OV \in \mathbb{OV}_R^{conf*} : (\nexists OV' \in \mathbb{OV}_R^{conf*} : OV' \subset OV)$$ ❍

Das nachstehende Lemma hält fest, dass $\mathbb{OV}_R^{conf*}$ eine gültige Abschätzung für $\mathbb{OV}_R^{conf}$ ist.

Lemma 4.2.5 ($\mathbb{OV}_R^{conf*}$ ist Abschätzung für $\mathbb{OV}_R^{conf}$)
Sei $R \in \mathbb{RW}$ ein Regelwerk. Dann gilt:

$$\forall OV \in \mathbb{OV}_R^{conf} : \exists OV^* \in \mathbb{OV}_R^{conf*} : OV \subseteq OV^* \qquad \text{(Beweis s. A.2.2, S. 297)} \ \square$$

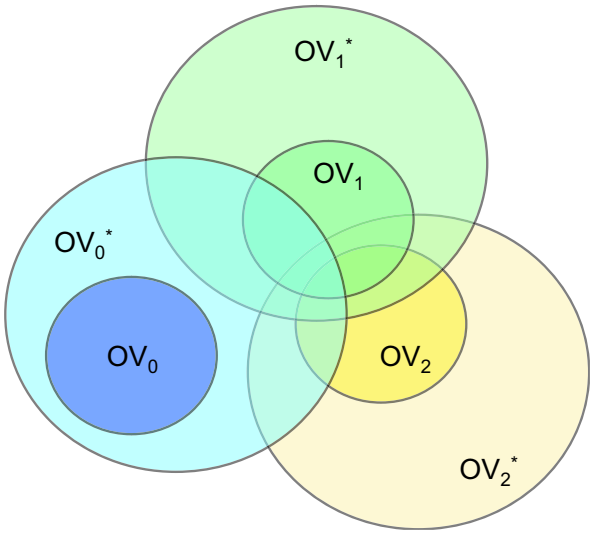


Abbildung 4.18: Graphische Darstellung des Verhältnisses der Mengen die in $\mathbb{OV}_R^{conf}$ und $\mathbb{OV}_R^{conf*}$ enthalten sind.

Abbildung 4.18 stellt die Situation graphisch dar. Die drei Mengen OV_0, OV_1 oder OV_2 stellen Elemente von $\mathbb{OV}_R^{conf}$ dar. Alle Objektvariablen die jeweils zu einer der drei Mengen gehören, können dementsprechend für ein festes Modell dasselbe Objekt erzeugen. Liegt eine Objektvariable in der Schnittmenge zwischen zwei dieser Mengen (z.B. zwischen OV_1 und OV_2), so können für eine gegebene, feste Menge von Quellmodellen entweder nur Objektvariablen der Menge OV_1 ein bestimmtes Objekt mehrfach erzeugen oder aber nur Objektvariablen aus der Menge OV_2.

Die Mengen OV_0^*, OV_1^* oder OV_2^* skizzieren das Ergebnis der Abschätzung $\mathbb{OV}_R^{conf*}$. Auch diese Mengen können sich überschneiden, jedoch enthält jede der Mengen mindestens eine der Mengen aus $\mathbb{OV}_R^{conf}$ vollständig. D.h. jede der Mengen von $\mathbb{OV}_R^{conf}$ ist mit Sicherheit in einer der Mengen zur Abschätzung von $\mathbb{OV}_R^{conf}$ enthalten (vergleiche Lemma 4.2.5).

Ermittlung der maximal möglichen Anzahl von Matches in Quellmodellen

Innerhalb dieses Abschnitts gilt es, Aussagen darüber machen zu können, wie oft ein Objekt durch eine Regel erzeugt werden kann, falls dessen Identität von den als fest angesehenen Modellvariablen abhängt. Um feststellen zu können, wie oft eine Quellmodellvariable höchstens in einem Modell matchen kann, wobei eine Menge von Objektvariablen immer auf dieselben Objekte matchen, während eine weitere Menge von Objektvariablen nie gleichzeitig auf dieselben Objekte matchen, wird die Funktion *maxmatch* eingeführt.

Weiterhin können durch diese Funktion Aussagen darüber getroffen werden, wie oft eine Modellvariable verschiedene Objekte erzeugt, während gleichzeitig immer wieder ein Objekt mit derselben

Identität geschaffen wird. Aussagen dieser Art werden für die Berechnung der maximal möglichen Zahl von Assoziationen benötigt, die durch eine Regel erzeugt werden können und alle die vom selben Zielobjekt ausgehen.

Definition 4.2.18 (*maxmatch*$(mv, \{ov_i, ..., ov_j\}, \{ov_k, ..., ov_l\})$**)**
Die Abbildung *maxmatch* sei folgendermaßen definiert:

$$maxmatch : \mathbb{MV} \times \mathscr{P}(\mathbb{OV}) \times \mathscr{P}(\mathbb{OV}) \to \mathbb{N}_0^\infty$$
$$maxmatch(mv, OV, \overline{OV}) \mapsto n \in \mathbb{N}_0^\infty \text{ mit } OV, \overline{OV} \subseteq mv|_{OVB}$$

Für eine Modellvariable *mv* und zwei Mengen von Objektvariablen *OV* und $\overline{OV}$ aus *mv* liefert die Funktion *maxmatch* die maximal mögliche Zahl von Matches der Modellvariable in einem beliebigen, zu $mv|_{mm}$ konformen Modell, für die gilt:

- Alle Objektvariablen aus *OV* matchen jeweils *immer* paarweise auf dieselben Objekte des Quellmodells.
- Alle Objektvariablen aus $\overline{OV}$ matchen jeweils *nie* paarweise auf dieselben Objekte des Quellmodells.

❍

Bemerkung: Für die folgenden Spezialfälle gilt:

$$maxmatch(mv, \bot, \overline{OV}) = 0$$
$$maxmatch(mv, \emptyset, \overline{OV}) = 0$$

Beispiel: Im Folgenden wird *maxmatch* anhand eines einfachen Beispiels erläutert. Zur besseren Übersichtlichkeit wurden nicht relevante Details wie Attribute, Rollennamen oder Identifikatorterme in Klassen und Modellvariablen des Beispiels weggelassen. Abbildung 4.19 zeigt ein Beispiel für ein Metamodell mit drei Klassen A, B, and C.

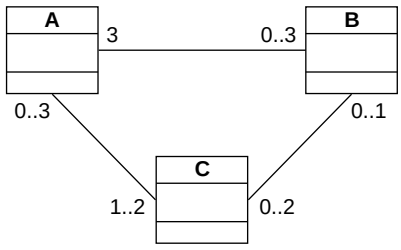


Abbildung 4.19: Das Metamodell *mm*

Die Abbildungen 4.20 und 4.21 zeigen zwei Modellvariablen mv_1 und mv_2 für die gilt:

$$mv_1|_{mm} = mv_2|_{mm} = mm$$

Im Beispiel wird für beide Modellvariablen davon ausgegangen, dass $\widehat{ov}$ die einzige als fest gekennzeichnete Objektvariable ist. Beim ersten Beispiel fällt es leicht zu sehen, dass es maximal zwei Möglichkeiten für die Wahl eines Objekts vom Typ C geben kann und nur eine für die Wahl eines Objekts

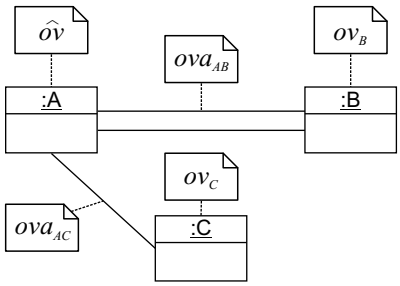


Abbildung 4.20: Die Modellvariable mv_1

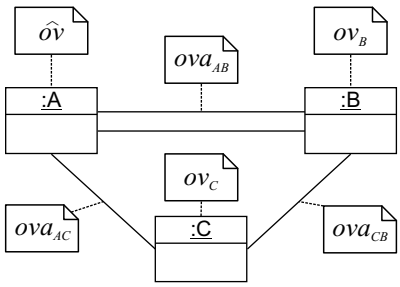


Abbildung 4.21: Die Modellvariable mv_2

vom Typ B, da dieses Objekt mit mindestens zwei von drei möglichen Assoziationen verbunden sein muss. Somit gilt:

$$maxmatch(mv_1, \{\widehat{ov}\}, \emptyset) = 2 \cdot 1 = 2$$

In dem Beispiel aus Abbildung 4.21 ist die Situation etwas komplizierter. Hier lässt sich das Ergebnis nicht ohne weiteres mit Hilfe kombinatorischer Mittel errechnen, da der durch die Modellvariable gebildete Graph nunmehr kein Baum ist. Dennoch lässt sich auch hier ermitteln:

$$maxmatch(mv_2, \{\widehat{ov}\}, \emptyset) = 2 \cdot 1 = 2$$

Abbildung 4.22 zeigt ein mögliches Modellfragment, in dem mv_2 zweimal matcht.

Speziell in Fällen, in denen mehrere Objektvariable, die nicht direkt verbunden sind, als „fest“ markiert sind, und in Fällen, in denen die Modellvariable keine Baumstruktur bildet, kann die Berechnung

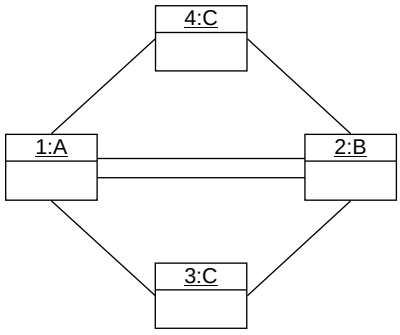


Abbildung 4.22: Beispiel für ein Modellfragment für *maxmatch*

von *maxmatch* sehr schwierig werden. ❍

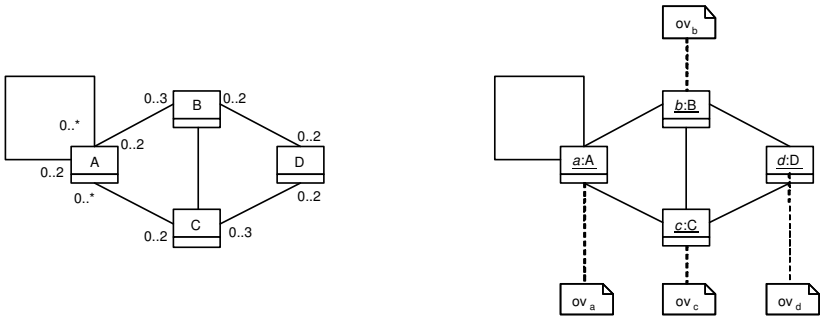


Abbildung 4.23: (a) Das Quellmetamodell mm_0 (b) Die Quellmodellvariable mv_0

Es wird nun anhand eines Beispiels dargestellt, wie sich die Funktion *maxmatch* in endlicher Zeit deterministisch berechnen lässt. Hierzu wird das in Abbildung 4.23 (a) dargestellte Quellmetamodell *mm* und die zugehörige Quellmodellvariable mv_0 aus Abbildung 4.23 (b) betrachtet. Da die Multiplizitätsuntergrenzen für die Berechnung von *maxmatch* irrelevant sind und um Konflikte, die aus der möglichen Nicht-Instanzierbarkeit des Quellmetamodells (siehe Abschnitt 3.2, S. 63 ff.) resultieren könnten, zu vermeiden, sind im Beispiel sämtliche Multiplizitätsuntergrenzen mit dem Wert 0 angegeben. Im Folgenden wird ein Verfahren skizziert mit dem sich für das angegebene Beispiel der Wert $maxmatch(mv_0, \{ov_a, ov_b\}, \emptyset)$ berechnen lässt.

Allgemein wird der Fall $maxmatch(mv_j, M, M')$ betrachtet, wobei gilt: $mm_i = mv|_{mm}$ und $M, M' \subseteq mv_j|_{OVB}$. Zunächst wird untersucht, wieviele Objekte in den Matches von mv_j bei festen Elementen aus M höchstens auftreten können. Hierzu lässt sich System aus Gleichungen und Ungleichungen formulieren. Das System enthält für jede Objektvariable *ov* aus mv_j eine Variable n_{ov}, die aussagt, wieviele Objekte in den Matches auftreten können, auf die die Objektvariable *ov* matcht.

$$GL: \quad \bigwedge_{ov' \in M} n_{ov'} = 1 \wedge \tag{4.24}$$

$$UGL: \quad \bigwedge_{\substack{ova' = (ovae'_0, ovae'_1) \in mv_i|_{OVA} \\ \text{mit } ovae'_0|_{ov} \neq ovae'_1|_{ov}}} n_{ovae'_0|_{ov}} \leq ub_{ovae'_0} \cdot n_{ovae'_1|_{ov}} \tag{4.25}$$

$$\text{mit } ovae'|_{ae}|_m = (lb_{ovae'}, ub_{ovae'})$$

(4.24) hält fest, dass Objektvariablen aus M immer auf dasselbe Objekt matchen. Dementsprechend kann ein solches Objekt nur einmal im gematchten Quellfragment enthalten sein. (4.25) reflektiert die Verhältnisse wieder, die sich aus den jeweiligen Multiplizitätsobergrenzen von Assoziationen ergeben. Somit lässt sich für jedes n_{ov} ein maximaler Wert berechnen. Dieser liegt in den meisten Fällen über dem Tatsächlichen. So wird beispielsweise der Umstand, dass die Multiplizitätsobergrenzen ggf. durch schon in der Modellvariablen „verbrauchte“ Objektvariablenassoziationen weiter eingeschränkt wird, nicht weiter berücksichtigt. Dies ist jedoch für das weitere Vorgehen nicht relevant, da lediglich nach einer oberen Grenze für die in der *maxmatch*-Berechnung involvierten Objekte gesucht wird. Erhält man für einen n_{ov} den Wert ∞, so ist davon auszugehen, dass *maxmatch* ebenfalls den Wert ∞ liefert.

Für das Beispiel ergibt sich somit das folgende System aus Gleichungen und Ungleichungen:

$$
\begin{array}{rlll}
GL: & n_{ov_a} = 1 & n_{ov_b} \leq 3 \cdot n_{ov_a} & n_{ov_c} \leq 4 \cdot n_{ov_b} \\
 & n_{ov_b} = 1 & n_{ov_b} \leq 1 \cdot n_{ov_c} & n_{ov_c} \leq 3 \cdot n_{ov_d} \\
UGL: & n_{ov_a} \leq \infty \cdot n_{ov_c} & n_{ov_b} \leq 2 \cdot n_{ov_d} & n_{ov_d} \leq 2 \cdot n_{ov_b} \\
 & n_{ov_a} \leq 2 \cdot n_{ov_b} & n_{ov_c} \leq 2 \cdot n_{ov_a} & n_{ov_d} \leq 2 \cdot n_{ov_c}
\end{array}
$$

Für Ungleichungssysteme dieser Art lässt sich für jedes n_{ov} ein maximaler Wert bestimmen. Im gegebenen Beispiel ist dies:

$$
\begin{array}{ll}
n_{ov_a} = 1 & n_{ov_c} = 2 \\
n_{ov_b} = 1 & n_{ov_d} = 2
\end{array}
$$

Nun lässt sich durch wiederholtes Einsetzen von Assoziationen in endlicher Zeit die Menge $MF_{max} = \{mf_0, \ldots, mf_m\}$ berechnen. Diese besteht aus sämtlichen obergrenzenkonformen Modellfragmenten mit maximalen Anzahlen von Assoziationen, die aus einem Objekt der Klasse *A*, einem der Klasse *B*, zwei der Klasse *C* und zwei der Klasse *D* bestehen und für die gilt:

$$\forall mf \{mf_0, \ldots, mf_m\} : \mathit{consistent}(mf, mm_i)$$

Assoziationen und Kardinalitäten von Assoziationen, welche in der Modellvariablen nicht vorkommen, brauchen hierbei nicht berücksichtigt zu werden. Somit ist sichergestellt, dass (für endliche n_{ov}-Werte) lediglich eine endliche Menge von Modellfragmenten erzeugbar ist. Das Ergebnis von $\mathit{maxmatch}(mv_j, M, \emptyset)$ ergibt sich dann als

$$\mathit{maxmatch}(mv_j, M, \emptyset) = \max\left\{|MFM(mf, mv_j)| : mf \in MF_{max}\right\}$$

Dementsprechend lässt sich das Ergebnis für den allgemeineren Fall $\mathit{maxmatch}(mv_j, M, M')$ ermitteln

$$
\begin{aligned}
\mathit{maxmatch}(mv_j, M, M') = \max\{\ & |MFM(mf, mv_j)| : mf \in MF_{max} \wedge \\
& \nexists mfm, mfm' \in MFM(mf, mv_j) \text{ mit } mfm \neq mfm' : \\
& \forall ov \in M' : mfm|_{match_o}(ov) = mfm'|_{match_o}(ov)\ \}
\end{aligned}
$$

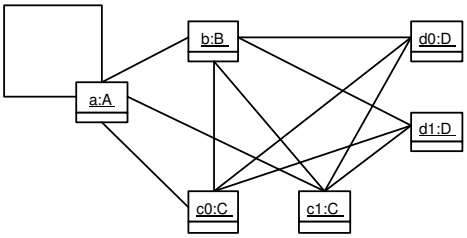


Abbildung 4.24: Das einzig mögliche Modellfragment auf das mv_0 bei festem ov_a und ov_b maximal oft matchen kann.

Im Beispiel enthält die Menge MF_{max} lediglich ein einzelnes maximales Modellfragment, d.h. alle anderen möglichen Modellfragmente sind Substrukturen und brauchen nicht weiter berücksichtigt zu

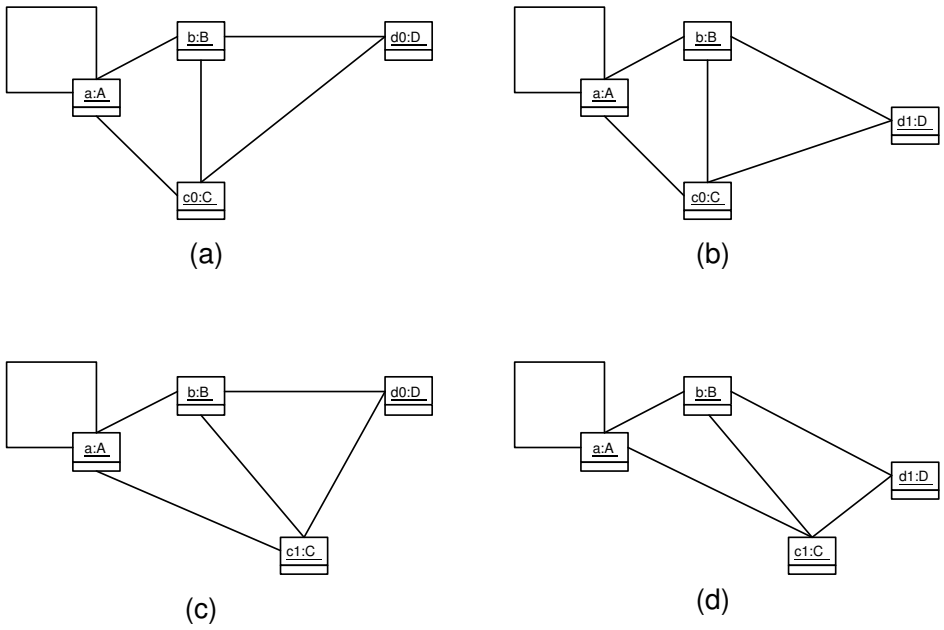


Abbildung 4.25: Die maximal vier möglichen Matches von mv_0 bei festem ov_a und ov_b in einem zu mm_0 konformen Modell.

werden. In der Allgemeinheit sind auch mehrere unterschiedlich strukturierte Modellfragmente möglich. Dieses maximale Modellfragment ist das in Abbildung 4.24 dargestellt. Wie sich leicht ersehen lässt, existieren für diesen Fall genau die in Abbildung 4.25 (a)-(d) dargestellten vier Modellfragment-Matches. Dementsprechend gilt:

$$maxmatch(mv_0, \{ov_a, ov_b\}, \emptyset) = 4$$

Durch das hier vorgestellte Verfahren lässt sich das Ergebnis der *maxmatch*-Funktion in endlicher Zeit bestimmen. Diese Möglichkeit ist vor allem für den automatisierten Nachweis der Metamodellkonformität von Regelwerken durch eine entsprechnede Werkzeugunterstützung bedeutend.

Abschätzung der Obergrenze für erzeugte, ausgehende Assoziationen

Um feststellen zu können, ob ein Regelwerk obergrenzenkonform ist, muss bekannt sein, wieviele Assoziationen eines gegebenen Typs maximal von einem Objekt des Zielmodells ausgehen können. Ist dieser Wert kleiner als die entsprechende Multiplizitätsobergrenze, so wird die Obergrenzenkonformität durch keine Assoziation des gegebenen Typs verletzt. Die Funktion *maxCard* liefert diesen Wert. In der Praxis lässt sich *maxCard* jedoch nicht ohne weiteres berechnen, so dass im weiteren Verlauf Techniken vorgestellt werden um das Ergebnis von *maxCard* nach oben abzuschätzen.

Definition 4.2.19 (*maxCard*(AE, ae_1, R))
Sei $R \in \mathbb{RW}$ ein Regelwerk mit dem Zielmetamodell mm^1_R. Weiter sei $AE \in mm^1_R|_{CA}$ und $ae_1 \in AE$. Dann ist die Funktion *maxCard* in folgender Weise definiert:

$$maxCard(AE, ae_1, R) \mapsto n \in \mathbb{N}^{\infty}_0 \text{ mit } ae_1 \in AE$$

$$maxCard(AE, ae_1, R) := \max_{M \in \mathbb{M}^0_R} \Big(\max_{o \in transform(M,R)|_{OB}} \Big(\sum_{\substack{oa \in transform(M,R)|_{OA} \text{ mit} \\ (oppositeEnd(AE,ae_1),o) \in oa|_{OAE}}} oa|_{card} \Big) \Big)$$

❍

$maxCard(AE, ae_1, R)$ liefert also die maximal mögliche Anzahl von erzeugten Assoziationen des Typs AE die von einem generierten Objekt ausgehen können. ae_1 spezifiziert das von o aus gesehen gegenüberliegende Ende der Assoziation.

Im Folgenden wird festgehalten, dass ein Regelwerk obergrenzenkonform ist, falls für alle im Zielmetamodell vorkommende Assoziationen der Wert *maxCard* kleiner oder gleich als die entsprechende Multiplizitätsobergrenze ist.

Satz 4.2.2 (*maxCard* zum Nachweis der Obergrenzenkonformität)
Sei $R \in \mathbb{RW}$ ein Regelwerk. Dann gilt:

$$\forall AE \in mm_R^1|_{CA}, ae \in AE \text{ mit } (lb, ub) := ae|_m : \; maxCard(AE, ae, R) \leq ub$$
$$\implies ubConform(R)$$

(Beweis s. A.2.2, S. 298) ❑

Somit kann $maxCard(AE, ae, R)$ verwendet werden, um zu zeigen, dass ein gegebenes Regelwerk obergrenzenkonform ist. Da sich *maxCard* jedoch in der Regel nicht direkt berechnen lässt, müssen stattdessen möglichst genaue Abschätzverfahren entwickelt werden, die es erlauben, eine obere Schranke für *maxCard* zu ermitteln.

Um den größtmöglichen Wert von aus einem Objekt ausgehenden Assoziationen, die von derselben Objektvariablenassoziation erzeugt wurden, zu bezeichnen, wird im weiteren die Abbildung *maxVarCard* eingeführt.

Definition 4.2.20 ($maxVarCard(ova, ae_1, r)$)
Sei $r \in \mathbb{R}$ eine Regel. Weiter sei $ova \in r|_{mv_1}|_{OVA}$ und $ae_1 \in ova|_{OVAT}$. Dann ist die Funktion *maxVarCard* in folgender Weise definiert:

$$maxVarCard : \mathbb{OVA} \times \mathbb{OVA}|_{OVAE}|_{ae} \times \mathbb{R} \to \mathbb{N}_0^+$$
$$maxVarCard(ova, ae_1, r) := \max_{m \in \mathbb{M}_{r|_{mv_0}|_{mm}}} \Big(\max_{o \in apply(m,r)|_{OB}} (numAssos(r, m, o, ova, ae_1)) \Big)$$ ❍

maxVarCard liefert die größtmögliche Anzahl von Assoziationen, die von einem Objekt ausgehen können und von derselben Objektvariablenassoziation erzeugt wurden. Dieser Wert gilt für beliebige Quellmodelle der Regel r.

Das nachfolgende Lemma 4.2.6 betrachtet die maximale Anzahl von aus einem durch R erzeugten Objekt ausgehenden Assoziationen eines Typs AE ($maxCard(AE, ae, R)$). Dieser Wert ist kleiner oder gleich jeder Summe über *maxVarCards*'s für nicht-redundante Objektvariablenassoziationen, an deren Enden Objektvariablen hängen die dasselbe Objekt erzeugen können.

Lemma 4.2.6 (*maxCard* und *maxVarCard*)
Es sei $R \in \mathbb{RW}$ ein Regelwerk, $AE_i \in mm_R^1$ und $ae_j \in AE_i$. Weiter sei $(lb, ub) := ae_j|_m$. Dann gilt:

$$maxCard(AE_i, ae_j, R)$$
$$\leq \max_{\substack{OV_k \in \mathbb{OV}_R^{conf} : \forall ov_m \in OV_k : \\ oppositeEnd(AE_i, ae_j)|_c \in types(ov_m|_{otv})}} \Big\{ \sum_{\substack{r_l \in R, ova_m \in r_l|_{mv_1}|_{OVA} : \exists ov_h \in OV_k : \\ \big(ov_h \in r_l|_{mv_1}|_{OVB} \\ \wedge (oppositeEnd(AE_i, ae_j), ov_h) \in ova|_{OVAE} \\ \wedge ova_m|_{OVAT} = AE_i \\ \wedge maxRelevant(R, r_l, ova_m)\big)}} maxVarCard(ova_m, ae_j, r_l) \Big\}$$

(Beweis s. A.2.2, S. 298) ❑

Die von *dependsOn* errechneten Werte für die Identität von *ov* sind eins-zu-eins abhängig von den Identifikatoren und Primärschlüsselattributen von *OV* (siehe Def. 4.1.6, S. 116). Somit ist sichergestellt, dass jedesmal wenn ein Objekt mit derselben Identität erzeugt wird, allen Elementen aus *OV* genau (paarweise) die *selbe* Menge von Quellobjekten matchen. Liefert $dependsOnId$ für eine Zielobjektvariable also eine Menge von Quellobjektvariablen als Ergebnis, so kann diese Objektvariable so oft das selbe Objekt erzeugen wie die Elemente aus *OV* während einer Regelanwendung auf die gleichen Objekte eines Zielmodells matchen können. Dieser Wert lässt sich mit der zuvor vorgestellten Abbildung *maxmatch* berechnen.

Im Folgenden wird die Funktion *ubVarCard* vorgestellt. Sie dient dazu abzuschätzen, wieviele aus demselben Objekt ausgehende Assoziationen höchstens von einer gegebenen Objektvariablenassoziation erzeugt werden können. Hierzu verwendet die Definition von *ubVarCard* die Abbildung *dependsOn*, die Aussagen darüber ermöglicht, wie oft Objekte an Enden von Assoziationen dieselben bzw. unterschiedlich sein können.

Definition 4.2.21 (*ubVarCard*(ova, ae_1, r))
Es sei $r \in \mathbb{R}$ eine Regel, $ova \in r|_{mv_1}|_{OVA}$ und $ae_1 \in ova|_{OVAT}$. Weiter sei $(lb, ub) := ae_1|_m$. Dann ist die Funktion *ubVarCard* wie folgt definiert:

$$ubVarCard : \mathbb{OVA} \times \mathbb{OVA}|_{OVAE}|_{ae} \times \mathbb{R} \to \mathbb{N}_0^+$$
$$ubVarCard(ova, ae_1, r) \mapsto n \text{ mit } ova \in r|_{mv_1}|_{OVA} \wedge ae_1 \in ova|_{OVAT}$$

ae_0, ov_0, ov_1 seien so gewählt, dass gilt:

$$ae_0 := oppositeEnd(ova|_{OVAT}, ae_1) \tag{4.26}$$
$$ova|_{OVAE} := \{(ae_0, ov_0), (ae_1, ov_1)\} \tag{4.27}$$

Dann gibt es zwei grundlegende Fälle für die Berechnung von *ubVarCard*:

Fall 1: $ov_0 = ov_1$ Abbildung 4.26 und Abbildung 4.27 zeigen die beiden möglichen Fälle für reflexive

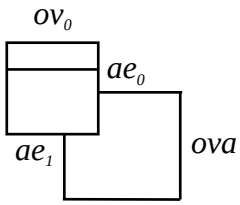


Abbildung 4.26: Reflexive Assoziation $(ov_0 = ov_1, ae_0 \neq ae_1)$

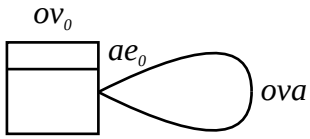


Abbildung 4.27: Symmetrische und reflexive Assoziation $(ov_0 = ov_1, ae_0 = ae_1)$

Objektvariablenassoziationen. Für beide Fälle gilt:

$$ubVarCard(ova, ae_1, r) := ova|_{cardv}$$

Fall 2: $ov_0 \neq ov_1$ Dieser Fall wird in Abbildung 4.28 veranschaulicht. Tabelle 4.2 definiert die Ergebnisse von *ubVarCard* für die 16 möglichen Fälle. ❍

Das nachstehende Lemma hält fest, dass *ubVarCard* eine gültige Abschätzung für die „ideale“ Funktion *maxVarVard* darstellt. D.h. *ubVarVard* liefert für jedes Ende einer Objektvariablenassoziation in einer Regel einen mindestens so großen Wert, wie der eigentlich gesuchte Wert von *maxVarCard*.

	$dependsOn(ov_0, r)$	$dependsOn(ov_1, r)$	$ubVarCard(ova, ae_1, r) :=$
(i)	$\emptyset$ (fest)	$\emptyset$	$ova\|_{cardv}$
(ii)	$\perp$ (beliebig)	$\emptyset$	$ova\|_{cardv}$
(iii)	$\emptyset \neq M_0 \subseteq r\|_{mv_0}\|_{OVB}$	$\emptyset$	$ova\|_{cardv}$
(iv)	$\emptyset$	$\perp$	∞
(v)	$\perp$	$\perp$	∞
(vi)	$\emptyset \neq M_0 \subseteq r\|_{mv_0}\|_{OVB}$	$\perp$	$maxmatch(r\|_{mv_0}, M_0, \emptyset) \cdot ova\|_{cardv}$
(vii)	$\emptyset$	$\emptyset \neq M_1 \subseteq r\|_{mv_0}\|_{OVB}$	∞
$(viii)$	$\perp$	$\emptyset \neq M_1 \subseteq r\|_{mv_0}\|_{OVB}$	∞
(ix)	$\emptyset \neq M_0 \subseteq r\|_{mv_0}\|_{OVB}$	$\emptyset \neq M_1 \subseteq r\|_{mv_0}\|_{OVB}$	$\begin{cases} ova\|_{cardv} & \text{falls } M_0 \overset{r\|_{mv_0}}{\rightsquigarrow} M_1 \\ maxmatch(& \\ \quad r\|_{mv_0}, M_0, M_1) & \text{sonst} \\ \cdot ova\|_{cardv} & \end{cases}$
(x)	$\Diamond$	$\emptyset$	$ova\|_{cardv}$
(xi)	$\Diamond$	$\perp$	$ova\|_{cardv}$
(xii)	$\Diamond$	$\emptyset \neq M_1 \subseteq r\|_{mv_0}\|_{OVB}$	$ova\|_{cardv}$
$(xiii)$	$\Diamond$	$\Diamond$	$ova\|_{cardv}$
(xiv)	$\emptyset$	$\Diamond$	∞
(xv)	$\perp$	$\Diamond$	∞
(xvi)	$\emptyset \neq M_0 \subseteq r\|_{mv_0}\|_{OVB}$	$\Diamond$	$maxmatch(r\|_{mv_0}, M_0, \emptyset) \cdot ova\|_{cardv}$

Tabelle 4.2: Definition von *ubVarCard* für den Fall $ov_0 \neq ov_1$

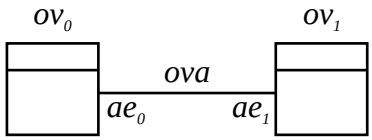


Abbildung 4.28: ov_0 und ov_1

Lemma 4.2.7 (*ubVarCard* ist Abschätzung für *maxVarCard*)
Es sei $r \in \mathbb{R}$ eine Regel, $ova \in r|_{mv_1}|_{OVA}$ und $ae_1 \in ova|_{OVAT}$. Dann gilt:

$$maxVarCard(ova, ae_1, r) \leq ubVarCard(ova, ae_1, r)$$ (Beweis s. A.2.2, S. 300) ❑

Der folgende Satz formuliert eine Verifikationstechnik für Obergrenzenkonformität. Es wird für jedes Klassenassoziationsende des Zielmetamodells die Summe aller Objektassoziationen, die von einem Objekt ausgehen können und die von einer relevanten Objektvariablenassoziation erzeugt wurden, gebildet. Hierbei werden jeweils nur Objektvariablenassoziationen gemeinsam betrachtet, welche von Objektvariablen ausgehen, die potentiell dasselbe Zielobjekt erzeugen können. Ist diese Summe für alle Klassenassoziationen kleiner als die jeweilige Multiplizitätsobergrenze und das Regelwerk anwendbar, so ist das Regelwerk auch obergrenzenkonform.

Satz 4.2.3 (Verifikationstechnik für Obergrenzenkonformität)
Sei $R \in \mathbb{RW}$ Regelwerk. Dann gilt:

(i) $\forall AE_i \in mm^1_R|_{CA}, ae_j \in AE_i$ mit $(lb, ub) := ae_j|_m$:

$$ub \geq \max_{\substack{OV_k \in \mathbb{OV}_R^{conf*}: \forall ov_m \in OV_k: \\ oppositeEnd(AE_i, ae_j)|_c \in types(ov_m|_{otv})}} \left\{ \sum_{\substack{r_l \in R, ova_m \in r_l|_{mv_1}|_{OVA}: \exists ov_h \in OV_k: \\ \left(ov_h \in r_l|_{mv_1}|_{OVB} \right. \\ \wedge (oppositeEnd(AE_i, ae_j), ov_h) \in ova|_{OVAE} \\ \wedge ova_m|_{OVAT} = AE_i \\ \left. \wedge relevant(R, r_l, ova_m) \right)}} ubVarCard(ova_m, ae_j, r_l) \right\} \wedge$$

(ii) R ist anwendbar.

$$\Longrightarrow ubConform(R)$$ (Beweis s. A.2.2, S. 303) ❑

Es bleibt anzumerken, dass die Eigenschaft, ob eine Regel Constraint-behaftet ist interessanter Weise keinen Einfluss auf die Obergrenzenkonformität hat. Erzeugt eine Regelanwendung aufgrund eines verletzten Constraints im Quellmodellfragment ein leeres Zielmodellfragment, so wird das nicht erzeugte Fragment dennoch in der *relevant*-Abschätzung berücksichtigt, da für beliebige Quellmodelle keine Aussagen getroffen werden können, wann ein solcher Constraint verletzt wird. Somit wird für den Nachweis der Obergrenzenkonformität immer vom schlechtest möglichen Fall ausgegangen: die Regel erzeugt immer ein Modellfragment, sobald die Modellvariable der linken Seite matcht. Folglich ist hierfür keine Sonderbehandlung für Constraint-behaftete Regeln erforderlich.

Eine mögliche Verbesserung der Abschätzung wäre es, Regeln mit Constraints, die niemals erfüllbar sind, generell nicht zu berücksichtigen, da solche Regeln immer nur leere Modellfragmente erzeugen. Da die Spezifikation solcher Regeln aber ohnehin keinen Sinn macht und um die Lesbarkeit des Formalismus zu erhöhen, wurde auf diese Optimierung verzichtet.

4.2.3 Verifikationstechniken für Untergrenzenkonformität

Im Verlauf dieses Abschnitts werden Verifikationstechniken vorgestellt, die es erlauben, für eine bestimmte Klasse von Regelwerken nachzuweisen, dass diese lediglich untergrenzenkonforme Modellfragmente erzeugen. Solche Regelwerke werden *untergrenzenkonforme* Regelwerke genannt. Die folgende Definition legt diesen Begriff formal fest.

Definition 4.2.22 (Untergrenzenkonforme Regelwerke(*lbConform*(R)))
Ein Regelwerk $R \in \mathbb{R}$ heißt genau dann *untergrenzenkonform*, wenn das im Folgenden definierte Prädikat *lbConform*(R) gilt:.

$$lbConform : \mathbb{RW} \rightarrow \mathbb{B}$$
$$lbConform(R) :\Leftrightarrow \forall M \in \mathbb{M}_R^0 : lbConform(transform(M,R), mm_R^1) \quad \text{❍}$$

Analog zur Untergrenzenkonformität von Regelwerken wird nun der Begriff Untergrenzenkonformität für eine einzelne Regel definiert. Eine Regel ist genau dann untergrenzenkonform, wenn ihre Anwendung nur Modellfragmente erzeugt, die untergrenzenkonform bezüglich des Zielmetamodells der Regel sind.

Definition 4.2.23 (Untergrenzenkonforme Regeln (*lbConform*(r)))
Eine Regel $r \in \mathbb{R}$ heißt genau dann *untergrenzenkonform*, wenn das im Folgenden definierte Prädikat *lbConform*(r) für die Regel erfüllt ist, wobei gilt:

$$lbConform : \mathbb{R} \rightarrow \mathbb{B}$$
$$lbConform(r) :\Leftrightarrow \forall m \in \mathbb{M}_{r|_{mv_0}|_{mm}} : lbConform(apply(m,r), r|_{mv_1}|_{mm}) \quad \text{❍}$$

Die Möglichkeit die Untergrenzenkonformität einzelner Regeln zu verifizieren ist mit Blick auf die Tatsache, dass Regelwerke kompositional bezüglich ihrer Untergrenzenkonformität sind, relevant. Im Verlauf dieses Abschnitts werden Verifikationstechniken für die Untergrenzenkonformität von Regeln vorgestellt und die Kompositionlität von Regelwerken bezüglich dieser Eigenschaft nachgewiesen. In den folgenden beiden Unterabschnitten wird zunächst jeweils eine Verifikationstechnik für diese Eigenschaft einer Regel vorgestellt.

Einfache Verifikationstechnik für die Untergrenzenkonformität einer Regel

An dieser Stelle wird eine sehr einfache und intuitive Technik für die Verifikation der Untergrenzenkonformität angeboten. Sie basiert darauf, dass jede Zielmodellvariable bereits die gemäß dem Zielmetamodell mindestens erforderliche Zahl ausgehender Objektvariablenassoziationen aus Objektvariablen aufweisen muss.

Im nachstehenden Lemma wird zunächst ein Objekt *o* eines erzeugten Modellfragments betrachtet. Das Lemma besagt, dass die Summe aller Kardinalitäten der von *o* ausgehenden Assoziationen eines gegebenen Typs gleich der Summe der Kardinalitäten der entsprechenden Objektvariablenassoziationen ist, die von der Objektvariablen ausgehen, von der *o* stammt.

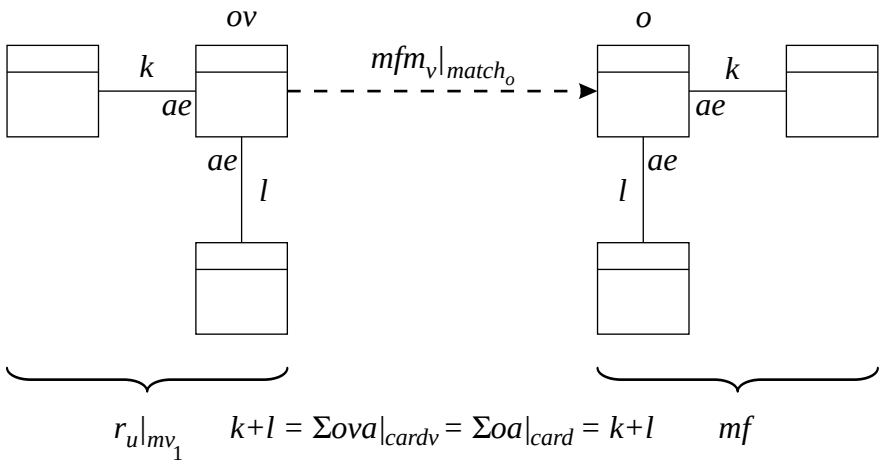


Abbildung 4.29: Summe der Kardinalitäten von ausgehenden Assoziationen

Lemma 4.2.8 (Durch eine Modellfragmenttransformation erzeugte Objektassoziationen)
Sei $r_u \in \mathbb{R}$ eine Regel mit dem Quellmetamodell mm_0 und $m \in \mathbb{M}_{mm_0}$ ein beliebiges aber festes Quellmodell. Für jedes Objekt o eines Zielmodellfragments mf, das durch eine Modellfragmenttransformation $mft(mfm_v, r_u)$ mit beliebigem aber gültigem (gemäß Definition 3.5.4) $mfm_v \in MFM(m, r|_{mv_0})$ entstanden ist, gilt:

$$\forall o \in mf|_{OVB}, AE \in mf|_{OA}|_{OAT}, ae \in AE \text{ mit } o|_{ot} \in types(AE|_c):$$
$$\exists ov \in r_u|_{mv_1}|_{OVB} \text{ mit } (i)\ o|_{ot} = ov|_{otv} \wedge$$
$$(ii) \sum_{\substack{ovae:ovae=(ae,ov) \\ \wedge ovae \in ova|_{OVAE} \\ \wedge ova|_{OVAT}=AE}} ova|_{cardv} = \sum_{\substack{oae:oae=(ae,o) \\ \wedge oae \in oa|_{OAE} \\ \wedge oa|_{OAT}=AE}} oa|_{card}$$

(Beweis s. A.2.2, S. 304) ❑

Abbildung 4.29 stellt den Sachverhalt exemplarisch dar. Hier entspricht die Summe der von *ov* ausgehenden Objektvariablenassoziationen genau der Summe der Assoziationen von *o* im erzeugten Zielmodellfragment.

Im folgenden Lemma wird die Summe der Kardinalitäten aller ausgehender Assoziationen eines Typs aus einem Zielobjekt *o* betrachtet. Diese ist sicher größer oder gleich der kleinsten Summe von Kardinalitäten der Objektvariablenassoziationen, die von Objektvariablen *ov* ausgehen, aus denen *o* hätte erzeugt werden können.

Lemma 4.2.9 (Durch eine Regelanwendung erzeugte Objektassoziationen)
Sei $r \in \mathbb{R}$ eine Regel und $m \in \mathbb{M}|_r|_{mv_0}|_{mm}$ ein beliebiges aber festes Quellmodell. Weiter sei

$$mf := apply(m, r)$$

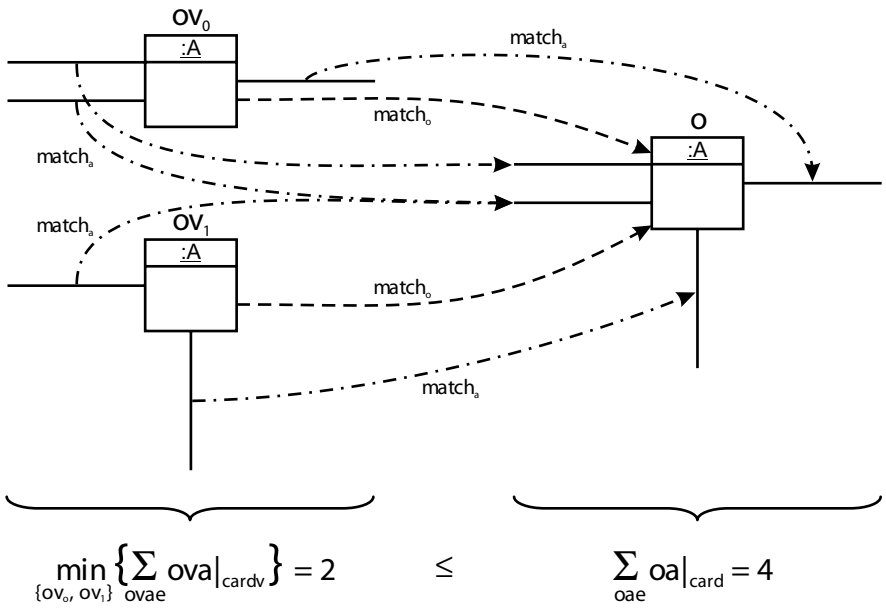


Abbildung 4.30: Beispiel für ein Objekt, das von zwei Objektvariablen erzeugt wurde

das durch Anwendung der Regel erzeugte Zielmodellfragment. Dann gilt:

$$\forall o \in mf|_{OB}, AE \in r|_{mm}|_{CA}, ae \in AE \text{ mit } ae|_c \in types(o|_t):$$
$$\overline{OV} := \{ov \in r|_{mv_1}|_{OVB} : ov|_{otv} = o|_{ot}\}$$
$$\min_{ov \in \overline{OV}} \left\{ \sum_{\substack{ova \in r|_{mv_1}|_{OVA}: \\ ova|_{OVAT} = AE \wedge \\ (ae,ov) \in ova|_{OVAE}}} ova|_{cardv} \right\} \leq \sum_{\substack{oa \in mf|_{OA}: \\ oa|_{OAT} = AE \wedge \\ (ae,o) \in oa|_{OAE}}} oa|_{card}$$

(Beweis s. A.2.2, S. 304) ❑

Abbildung 4.30 veranschaulicht die Aussage des Lemmas graphisch. Die linke Seite der Abbildung zeigt zwei Objektvariablen einer Regel mit einer Reihe ausgehender Objektvariablenassoziationen desselben Typs. Auf der rechten Seite ist ein Objekt mit seinen ausgehenden Objektassoziationen zu sehen, welches im Verlauf der Regelanwendung von diesen beiden Objektvariablen erzeugt wurde.

Im Beispiel wurden zwei aus Objektvariablenassoziationen erzeugte Objektassoziationen zu einer Objektassoziation zusammengeführt. Das Objekt selbst ist ebenfalls das Ergebnis des Zusammenführens zweier Modellfragmente. Unter dem Objekt und der Objektvariable sind die Summen der Kardinalitäten aus Lemma 4.2.9 zu sehen.

Neben Modellfragmenten können auch Modellvariablen Multiplizitätsuntergrenzen ihres jeweiligen Metamodells verletzen. Das Prädikat *varLbConform* gibt an ob dies für eine Modellvariable der Fall ist.

Definition 4.2.24 (*varLbConform*(mv))
varLbConform(mv) ist ein Prädikat, so dass gilt:

$$
\begin{aligned}
&\mathit{varLbConform} : \mathbb{MV} \rightarrow \mathbb{B} \\
&\mathit{varLbConform}(mv) :\Leftrightarrow \forall ov \in mv|_{OVB}, AE \in mv|_{mm}|_{CA}, ae \in AE \\
&\qquad \text{mit } ae|_c \in \mathit{types}(ov|_{otv}), (lb, ub) := \mathit{oppositeEnd}(AE, ae)|_m : \\
&\qquad\qquad lb \leq \sum_{\substack{ova \in mv|_{OVA}: \\ ova|_{OVAT} = AE \\ \wedge (ae, ov) \in ova|_{OVAE}}} ova|_{cardv}
\end{aligned}
$$

❍

varLbConform(mv) gilt genau dann, wenn die Summe aller Kardinalitäten ausgehender Objektvariablenassoziationen eines Typs für jede Objektvariable der Modellvariable *mv* größer oder gleich der Multiplizitätsuntergrenze der entsprechenden Klassenassoziation im Metamodell von *mv* ist.

Der nachfolgende Satz besagt, dass jede Regel, für deren Zielmodellvariable *varLbConform* gilt, untergrenzenkonform ist.

Satz 4.2.4 (Einfache Verifikationstechnik für die Untergrenzenkonformität einer Regel)
Sei $r \in \mathbb{R}$ eine anwendbare Regel. Dann gilt:

$$\mathit{varLbConform}(r|_{mv_1}) \Rightarrow \mathit{lbConform}(r)$$ (Beweis s. A.2.2, S. 305) ❑

Informell ausgedrückt lässt sich die Untergrenzenkonformität einer Regel anhand der folgenden beiden Schritte leicht per Augenschein verifizieren:

1. Für alle Objektvariablen in Zielmodellvariablen der Regel ist die Untergrenze für ausgehende Assoziationen im Zielmetamodell festzustellen.

2. Für jede dieser Objektvariablen wird überprüft, ob die Summe der ausgehenden Objektvariablenassoziationen des jeweiligen Assoziationstyps größer oder gleich als die in Punkt (1) festgestellte Untergrenze ist. Ist dies immer der Fall, so ist die Regel untergrenzenkonform.

Somit kann ein Regeldesigner leicht sicherstellen, dass seine Regeln untergrenzenkonform sind, indem er jede Objektvariable mit der benötigten Mindestanzahl von ausgehenden Assoziationen versieht. In einigen Fällen führt dies jedoch zu relativ redundanten und unübersichtlichen Regeln. Daher wird im Folgenden eine mächtigere Verifikationstechnik zur Überprüfung der Untergrenzenkonformität von Regeln vorgestellt.

Erweiterte Verifikationstechnik für die Untergrenzenkonformität einer Regel

Oftmals ist es nicht möglich, sämtliche vom Metamodell geforderten Assoziationen in einer Regel anzugeben. Hat eine ausgehende Klassenassoziation z.B. eine Multiplizitätsuntergrenze größer als 1, so soll verifiziert werden können, dass eine Regel, in welcher eine entsprechende Objektvariablenassoziation nur einmal vorkommt, dennoch die verlange Anzahl ausgehender Assoziationen an jedem Objekt erzeugt. Ein Beispiel für eine solche Regel findet sich auch in Abbildung 5.24 von Abschnitt 5.3 (S. 207).

Aus diesem Grund wird innerhalb dieses Abschnitts einer erweiterte Verifikationstechnik vorgestellt, die es erlaubt, auch für Regeln, für die *varLbConform* nicht gilt, nachzuweisen, dass diese untergrenzenkonform sind.

Analog zu *maxCard* aus Abschnitt 4.2.2 wird nun zunächst die Funktion *minCard* eingeführt. Diese liefert die minimale Anzahl von ausgehenden Assoziationen eines Typs aus demselben Objekt, welche durch die Anwendung einer gegebenen Regel erzeugt wurden. Im Gegensatz zu *maxCard* bezieht sich *minCard* jedoch lediglich auf eine einzelne Regel und nicht auf ein Regelwerk.

Definition 4.2.25 (*minCard*(AE, ae_1, r))
Seien $mm_0, mm_1 \in \mathbb{MM}$ Metamodelle und $r \in \mathbb{R}_{mm_0,mm_1}$ eine Regel. Weiter sei $AE \in mm_1|_{CA}$ eine Klassenassoziation ihres Zielmetamodells und $ae_1 \in AE$ eines seiner Enden. Dann gilt:

$$minCard(AE, ae_1, r) \mapsto n \in \mathbb{N}_0^\infty$$
$$minCard(AE, ae_1, r) := \min_{m \in \mathbb{M}_{mm_0}} \Big\{ \min_{o \in apply(m,r)|_{OB}} \Big\{ \sum_{\substack{ova \in r|_{mv_1}|_{OVA}: \\ ova|_{OVAT} = AE}} numAssos(r, m, o, ova, ae_1) \Big\} \Big\}$$ ❍

minCard gibt also an, wieviele Instanzen von Assoziationen eines Typs mindestens an jedem passenden Ausgangsobjekt generiert werden. Ist dieser Wert für jede Assoziation mit Sicherheit größer als die jeweilige Multiplizitätsuntergrenze der jeweiligen Assoziation, so sind die erzeugten Modellfragmente untergrenzenkonform. Da sich im allgemeinen Fall der Wert für *minCard* nicht immer exakt berechnen lässt, gilt es im weiteren eine geeignete, berechenbare Abschätzung für *minCard* zu finden, welche mit Sicherheit immer *kleinere* Werte als *minCard* liefert. Im Folgenden wird *lbCard* als eine solche Abschätzung eingeführt und die Gültigkeit dieser Abschätzung bewiesen (siehe Lemma 4.2.12, S. 164).

Im nachstehenden Lemma wird zunächst festgehalten, dass in einem, durch eine Regelanwendung erzeugten, Modellfragment jedes Objekt mindestens die Anzahl ausgehender Assoziationen eines Typs hat, die durch *minCard* festgelegt ist:

Lemma 4.2.10 (*minCard* ist untere Grenze für die Zahl ausgehender Objektassoziationen)
Seien $mm_0, mm_1 \in \mathbb{MM}$ Metamodelle und $r \in \mathbb{R}_{mm_0,mm_1}$ eine Regel. Weiter sei $AE \in mm_1|_{CA}$ eine Klassenassoziation ihres Zielmetamodells und $ae_1 \in AE$ eines seiner Enden. Für ein beliebiges aber festes Quellmodell $m \in \mathbb{M}_{mm_0}$ sei

$$mf := apply(m, r)$$

das durch die Regel r erzeugte Zielmodellfragment. Dann gilt:

$$\forall o \in mf|_{OB} : \sum_{\substack{oa \in mf|_{OA}: \\ oa|_{oat} = AE \ \wedge \\ (ae_1, o) \in oa|_{OAE}}} oa|_{card} \geq minCard(AE, ae_1, r)$$

(Beweis s. A.2.2, S. 305) ❑

Das nun definierte *minVarCard* liefert die minimal mögliche Anzahl von Assoziationen, die von einem Objekt ausgehen und die von derselben Objektvariablenassoziation *ova* erzeugt wurden. Dieser Wert gilt für beliebige Quellmodelle.

Definition 4.2.26 (*minVarCard*(ova, ae_1, r))
Seien $mm_0, mm_1 \in \mathbb{MM}$ Metamodelle und $r \in \mathbb{R}_{mm_0,mm_1}$ eine Regel. Weiter sei $ova \in r|_{mv_1}|_{OVA}$ eine Objektvariablenassoziation ihrer rechten Regelseite und $ae_1 \in ova|_{OVAT}$ ein zu *ova* passendes Klassenassoziationsende. Dann gilt:

$$minVarCard(ova, ae_1, r) \mapsto n \in \mathbb{N}_0^\infty$$
$$minVarCard(ova, ae_1, r) := \min_{m \in \mathbb{M}_{mm_0}} \Big\{ \min_{o \in apply(m,r)|_{OB}} \{ numAssos(r, m, o, ova, ae_1) \} \Big\}$$ ❍

Mit Hilfe von *minVarCard* ist bereits eine Abschätzung für *minCard* möglich. Die minimale Zahl, der von einer Objektvariablenassoziation erzeugten Assoziationen eines Typs an einem Objekt, ist sicherlich kleiner als die minimale Zahl aller Assoziationen dieses Typs an einem Objekt, die von derselben Objektvariablenassoziation erzeugt wurden. Lemma 4.2.11 hält diese Eigenschaft fest.

Lemma 4.2.11 (Abschätzung von *minCard* mit *minVarCard*)
Seien $mm_0, mm_1 \in \mathbb{MM}$ Metamodelle, $r \in \mathbb{R}_{mm_0,mm_1}$ eine Regel, $AE \in mm_1|_{CA}$ eine Klassenassoziation ihres Zielmetamodells und $ae_1 \in AE$ ein Klassenassoziationsende der Klassenassoziation AE. Dann gilt:

$$minCard(AE, ae_1, r) \geq \min_{\substack{ova \in r|_{mv_1}|_{OVA}: \\ ova|_{OVAT} = AE}} \{minVarCard(ova, ae_1, r)\}$$

(Beweis s. A.2.2, S. 305) ❑

minVarCard erlaubt zwar eine pessimistische Abschätzung von *minCard*, d.h. es lassen sich mit Sicherheit kleinere Werte berechnen, allerdings ist *minVarCard* im allgemeinen Fall selbst nicht berechenbar. Daher wird die Funktion *lbVarCard* (*lb* steht hierbei für „lower bounds“) als Abschätzung für *minVarCard* eingeführt. Analog zur Funktion *ubVarCard* aus Abschnitt 4.2.2 wird hierzu zunächst die Funktion *minmatch* definiert.

Definition 4.2.27 (*minmatch*$(mv, \{ov_i, \ldots, ov_j\}, \{ov_k, \ldots, ov_l\})$)
Die Abbildung *minmatch* sei wie folgt definiert:

$$minmatch : \mathbb{MV} \times \mathscr{P}(\mathbb{OV}) \times \mathscr{P}(\mathbb{OV}) \to \mathbb{N}_0^\infty$$
$$minmatch(mv, OV, \overline{OV}) \mapsto n \in \mathbb{N}_0^\infty \text{ mit } OV, \overline{OV} \subseteq mv|_{OVB}$$

Für eine Modellvariable *mv* und zwei Mengen von Objektvariablen *OV* und $\overline{OV}$ aus *mv* liefert die Funktion *minmatch* die minimal mögliche Zahl von Matches der Modellvariable in einem beliebigen, zu $mv|_{mm}$ konformen Modell, falls mindestens ein Match dieser Modellvariable existiert, für die gilt:

- Alle Objektvariablen aus *OV* matchen jeweils *immer* paarweise auf dieselben Objekte des Quellmodells.
- Alle Objektvariablen aus $\overline{OV}$ matchen jeweils *nie* paarweise auf dieselben Objekte des Quellmodells. ❍

Bemerkung: Für die folgenden Spezialfälle gilt:

$$minmatch(mv, \bot, \overline{OV}) = 0$$
$$minmatch(mv, \emptyset, \overline{OV}) = 0$$

Die nun eingeführte Funktion *lbVarCard* wird verwendet, um (pessimistisch) abzuschätzen, wie viele vom selben Objekt ausgehende Assoziationen, die von derselben Objektvariablenassoziation stammen, mindestens im Zielmodell generiert werden können. Hierfür stützt sich die Definition von *lbVarCard* auf die Abbildung *dependsOn*, die Aussagen darüber macht, wie oft Objekte an den Enden von Assoziationen identisch bzw. verschieden sein können.

Definition 4.2.28 (*lbVarCard*(ova, ae_1, r))
Es sei $r \in \mathbb{R}$ eine Regel, $ova \in r|_{mv_1}|_{OVA}$ und $ae_1 \in ova|_{OVAT}$. Weiter sei $(lb, ub) := ae_1|_m$. Dann ist die Funktion *lbVarCard* wie folgt definiert:

$$lbVarCard : \mathbb{OVA} \times \mathbb{OVA}|_{OVAE}|_{ae} \times \mathbb{R} \to \mathbb{N}_0^+$$
$$lbVarCard(ova, ae_1, r) \mapsto n \in \mathbb{N}_0^\infty \text{ mit } ova \in r|_{mv_1}|_{OVA} \wedge ae_1 \in ova|_{OVAT}$$

Sei ae_0, ov_0, ov_1 so gewählt, dass gilt

$$ae_0 = oppositeEnd(ova|_{OVAT}, ae_1) \tag{4.28}$$
$$ova|_{OVAE} = \{(ae_0, ov_0), (ae_1, ov_1)\} \tag{4.29}$$

Fall 1: $constrainted(r)$

$$lbVarCard(ova, ae_1, r) := ova|_{cardv}$$

Fall 2: $\neg constrainted(r)$

Fall 2.1: $ov_0 = ov_1$ Die Abbildungen 4.31 und 4.32 zeigen die beiden möglichen Fälle für die

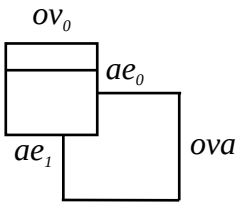


Abbildung 4.31: Reflexive Assoziation $(ov_0 = ov_1, ae_0 \neq ae_1)$

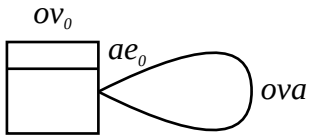


Abbildung 4.32: Symmetrische und reflexive Assoziation $(ov_0 = ov_1, ae_0 = ae_1)$

Assoziation *ova*.

$$lbVarCard(ova, ae_1, r) := ova|_{cardv}$$

Fall 2.2: $ov_0 \neq ov_1$

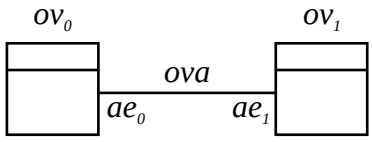


Abbildung 4.33: ov_0 und ov_1

Die Situation ist in Abbildung 4.33 illustriert. Tabelle 4.3 enthält die Definition von *lbVarCard* für die 16 möglichen Fälle.

❍

Lemma 4.2.12 besagt, dass das berechenbare *lbVarCard* eine gültige Abschätzung für das Ergebnis von *minVarCard* ist, d.h. es liefert immer kleinere oder gleichgroße Werte wie *minVarCard*:

	$dependsOn(ov_0, r)$	$dependsOn(ov_1, r)$	$lbVarCard(ova, ae_1, r) :=$
(i)	$\emptyset$ (fest)	$\emptyset$	$ova\|_{cardv}$
(ii)	$\perp$ (beliebig)	$\emptyset$	$ova\|_{cardv}$
(iii)	$\emptyset \neq M_0 \subseteq r\|_{mv_0}\|_{OVB}$	$\emptyset$	$ova\|_{cardv}$
(iv)	$\emptyset$	$\perp$	$ova\|_{cardv}$
(v)	$\perp$	$\perp$	$ova\|_{cardv}$
(vi)	$\emptyset \neq M_0 \subseteq r\|_{mv_0}\|_{OVB}$	$\perp$	$ova\|_{cardv}$
(vii)	$\emptyset$	$\emptyset \neq M_1 \subseteq r\|_{mv_0}\|_{OVB}$	$minmatch(r\|_{mv_0}, \emptyset, M_1) \cdot ova\|_{cardv}$
$(viii)$	$\perp$	$\emptyset \neq M_1 \subseteq r\|_{mv_0}\|_{OVB}$	$ova\|_{cardv}$
(ix)	$\emptyset \neq M_0 \subseteq r\|_{mv_0}\|_{OVB}$	$\emptyset \neq M_1 \subseteq r\|_{mv_0}\|_{OVB}$	$minmatch(r\|_{mv_0}, M_0, M_1) \cdot ova\|_{cardv}$
(x)	$\diamondsuit$	$\emptyset$	$ova\|_{cardv}$
(xi)	$\diamondsuit$	$\perp$	$ova\|_{cardv}$
(xii)	$\diamondsuit$	$\emptyset \neq M_1 \subseteq r\|_{mv_0}\|_{OVB}$	$ova\|_{cardv}$
$(xiii)$	$\diamondsuit$	$\diamondsuit$	$ova\|_{cardv}$
(xiv)	$\emptyset$	$\diamondsuit$	$ova\|_{cardv}$
(xv)	$\perp$	$\diamondsuit$	$ova\|_{cardv}$
(xvi)	$\emptyset \neq M_0 \subseteq r\|_{mv_0}\|_{OVB}$	$\diamondsuit$	$minmatch(r\|_{mv_0}, M_0, \emptyset) \cdot ova\|_{cardv}$

Tabelle 4.3: Definition von *lbVarCard* für den Fall $ov_0 \neq ov_1$

Lemma 4.2.12 (Abschätzung von *minVarCard* mit *lbVarCard*)
Es sei $r \in \mathbb{R}$ eine Regel, $ova \in r|_{mv_1}|_{OVA}$ und $ae \in ova|_{OVAT}$. Dann gilt:

$$minVarCard(ova, ae, r) \geq lbVarCard(ova, ae, r)$$ ❑

Der Beweis von Lemma 4.2.12 wird an dieser Stelle dem Leser überlassen, da die hierzu notwendigen Überlegungen denen zum Beweis von Lemma 4.2.7 sehr ähnlich sind. Interessant ist hier lediglich der Fall 1, bei dem die Regel r Constraint-behaftet ist. In diesem Fall lässt sich keine gesicherte Aussage darüber machen, wie oft die Regel nichtleere Zielmodellfragmente erzeugt, da das Ergebnis der Modellfragmenttransformation von Attributwerten im Quellmodell abhängt. Erzeugt die Regel niemals ein nichtleeres Modellfragment, so werden keine Objekte angelegt, für welche die Untergrenzenmultiplizität einer ausgehenden Assoziation verletzt werden könnte. Um also eine minimale Anzahl ausgehender Assoziationen zu erzeugen, muss die Regel genau einmal matchen, was der Aussage entspricht, dass die erzeugten Assoziation genau die Kardinalität $ova|_{cardv}$ hat.

Definition 4.2.29 führt die Funktion *lbCard* ein, die im Weiteren als Abschätzung für die Funktion *minCard* dient.

Definition 4.2.29 (*lbCard*(AE, ae_1, r))
Es sei $r \in \mathbb{R}$ eine Regel, $AE \in r|_{mv_1}|_{mm}|_{CA}$ und $ae_1 \in ova|_{OVAT}$. Weiter sei $(lb, ub) := ae_1|_m$ und $ae_0 := oppositeEnd(AE, ae_1)$. Dann ist die Funktion *lbCard* wie folgt definiert:

$$lbCard : \mathbb{AE} \times \mathbb{AE}|_{ae} \times \mathbb{R} \to \mathbb{N}_0^\infty$$
$$lbCard(AE, ae_1, r) :=$$
$$\begin{cases} 0 & \text{falls } \exists ov \in r|_{mv_1}|_{OVB} \text{ mit } ae_0|_c \in types(ov|_{otv}) : \\ & \quad \nexists ova \in r|_{mv_1}|_{OVA} : \ (ae_0, ov) \in ova|_{OVAE} \\ \min\limits_{\substack{ova \in r|_{mv_1}|_{OVB}: \\ ova|_{OVAT} = AE}} \{lbVarCard(ova, ae_1, r)\} & \text{sonst} \end{cases}$$ ❍

Der erste Fall von Definition 4.2.29 beschreibt die Situation, in der auf der rechten Regelseite eine Objektvariable *ov* existiert, deren Typ der gleiche ist, wie der von dem Assoziationen des Typs *AE* ausgehen. Weiterhin existiert in der Zielmodellvariablen $r|_{mv_1}$ keine Objektvariablenassoziation dieses Typs. In diesem Fall wird davon ausgegangen, dass keine ausgehende Assoziation an dem aus der Objektvariablen erzeugten Objekt entsteht.

Das nachfolgende Lemma zeigt, dass die Funktion *lbCard* eine gültige Abschätzung für *minCard* ist. Dementsprechend liefert *lbCard* für die gleiche Eingabe immer Werte die kleiner oder gleich dem Ergebnis von *minCard* sind.

Lemma 4.2.13 (*lbCard* ist Abschätzung für *minCard*)
Sei $r \in \mathbb{R}$ eine Regel, $AE \in r|_{mv_1}|_{mm}|_{CA}$ eine Klassenassoziation des Zielmetamodells und *ae* eines seiner Assoziationsenden. Dann gilt:

$$minCard(AE, ae, r) \geq lbCard(AE, ae, r)$$ (Beweis s. A.2.2, S. 306) ❑

Mit Hilfe der Abschätzung *lbCard*, welche eine untere Schranke für den Wert von *minCard* liefert, lässt sich nun die Untergrenzenkonformität einer einzelnen Regel nachweisen:

Satz 4.2.5 (Verifikationstechnik für die Untergrenzenkonformität einer Regel)
Sei $r \in \mathbb{R}$ eine Regel mit einer Zielmodellvariable $mv := r|_{mv_1}$ und dem Zielmetamodell $mm := mv|_{mm}$. Dann gilt:

$$\begin{aligned}&\forall AE \in mm|_{CA}, ae \in AE \text{ mit } ae|_c \in \bigcup_{class \in mv|_{OVB}|_{otv}} types(class), (lb, ub) := ae|_m : \\ &\quad lbCard(AE, ae, r) \geq lb \\ &\implies lbConform(r)\end{aligned}$$

(Beweis s. A.2.2, S. 306) ❑

$$\begin{aligned} & minCard(AE, ae_1, r) \\ \geq & \qquad \text{(Lemma 4.2.11)} \\ & \min_{ova}(minVarCard(ova, ae_1, r)) \\ \geq & \qquad \text{(Lemma 4.2.12)} \\ lbCard(AE, ae_1, r) = & \min_{ova}(lbVarCard(ova, ae_1, r)) \qquad \text{(Verif. Techn.)} \\ \geq & \qquad \text{(Satz 4.2.6)} \\ & lb \end{aligned}$$

Abbildung 4.34: Beweiskette für die Verifikationstechnik für Untergrenzenkonformität

Abbildung 4.34 skizziert informell die Beweiskette für die erweiterte Verifikationstechnik für die Untergrenzenkonformität einer Regel. *minCard* liefert für eine Regel und einen Assoziationstyp die minimal mögliche Anzahl von generierten Assoziationen, die von einem Objekt ausgehen. Das Minimum über ale *minVarCard*s ist bereits eine ideale Abschätzung. *lbVarCard* ist eine leicht zu berechnende (pessimistische) Abschätzung für *minVarCard*, d.h. *lbVarCard* liefert immer einen kleineren oder den gleichen Wert wie *minVarCard*. Ist also dieser Wert mindestens immer so groß wie die Untergrenze der jeweiligen Assoziationsmultiplizität, so ist die Regel untergrenzenkonform.

Es stehen nun zwei Techniken zur Verfügung um die Untergrenzenkonformität einer Regel zu beweisen. Diese lassen sich beide in der Verifikationstechnik zum Nachweis der Untergrenzenkonformität eines Regelwerks kombinieren, wie im nachstehenden Abschnitt gezeigt wird

Verifikationstechnik für die Untergrenzenkonformität eines Regelwerks

Satz 4.2.6 hält fest, dass Regelwerke, welche lediglich aus Untergrenzenkonformen Regeln bestehen untergrenzenkonform sind:

Satz 4.2.6 (Verifikationstechnik für die Untergrenzenkonformität eines Regelwerks)
Sei $R \in \mathbb{RW}$ ein anwendbares Regelwerk. Dann gilt:

$$\forall r \in R : lbConform(r) \implies lbConform(R)$$

(Beweis s. A.2.2, S. 306) ❑

Um die Untergrenzenkonformität eines Regelwerks nachzuweisen, reicht es also die Untergrenzenkonformität jeder einzelnen Regel des Regelwerks nachzuweisen. Für diesen Nachweis kann nun wahlweise entweder Satz 4.2.4 oder Satz 4.2.5 herangezogen werden.

Das folgende Lemma besagt, dass durch die Komposition untergrenzenkonformer Regelwerke wieder ein untergrenzenkonformes Regelwerk entsteht, falls dieses Regelwerk anwendbar ist. Das Lemma wird im weiteren Verlauf für den Nachweis der Untergrenzenkonformität von Rewrite-Regeln (siehe Abschnitt 3.6) benötigt werden.

Lemma 4.2.14 (Regelwerke sind kompositional bezüglich ihrer Untergrenzenkonformität)
Seien $R_0, R_1 \in \mathbb{RW}$ Regelwerke. Dann gilt:

$$R_0 \cup R_1 \text{ ist anwendbar } \wedge \mathit{lbConform}(R_0) \wedge \mathit{lbConform}(R_1)$$
$$\Longrightarrow \mathit{lbConform}(R_0 \cup R_1)$$

(Beweis s. A.2.2, S. 306) ❑

4.2.4 Verifikationstechnik für Metamodellkonformität

Der Nachweis seiner Ober- und Untergrenzenkonformität genügt, um zu zeigen, dass ein Regelwerk Metamodellkonform gemäß Definition 4.2.1 ist. Der nachfolgende Satz fasst dieses Ergebnis zusammen.

Satz 4.2.7 (Verifikationstechnik zum Nachweis von Metamodellkonformität)
Ein Regelwerk $R \in \mathbb{RW}$ ist metamodellkonform (d.h. $\mathit{mmConform}(R)$), falls gilt:

(i) $\mathit{ubConform}(R) \wedge$

(ii) $\mathit{lbConform}(R)$

(Beweis s. A.2.2, S. 307) ❑

Bemerkung: Sowohl $\mathit{lbConform}(R)$ als auch $\mathit{ubConform}(R)$ implizieren bereits, dass R anwendbar ist.

In Anhang E findet sich ein automatisch generierter Beweis der Metamodellkonformität des in Abschnitt 3.3 vorgestellten Beispielregelwerks R.

Offenbar stellt jedes metamodellkonforme Regelwerk eine metamodellkonforme Modelltransformationsspezifikation gemäß Definition 2.2.2 (Seite 40) dar.

4.2.5 Richtlinien zum Erstellen von BOTL-Spezifikationen

In der Praxis hat sich gezeigt, dass die vorliegenden Verifikationstechniken in den allermeisten Fällen ausreichen, um die Metamodellkonformität von BOTL-Spezifikationen nachzuweisen. Durch eine intelligente Modellierung der Transformationsregeln, kann die Klasse der Transformationen, deren Metamodellkonformität durch diese Techniken nachgewiesen werden, jedoch noch weiter erhöht werden. Nachstehend findet sich eine Aufzählung von Richtlinien für die Erstellung von BOTL-Spezifikationen, welche die Erstellung metamodellkonformer Regelwerke erleichtern.

- Ist es im Verlauf einer Transformation notwendig Quellobjekte gleichen Typs zu unterschieden, so kann die Unterscheidung entweder durch den Kontext des Objektes (die Assoziationen zu anderen Objekten, über die das Objekt verfügt) oder aber über seine Attributwerte erfolgen. Da BOTL in der bisherigen Form über keinen negativen Kontext in Quellmodellvariablen verfügt, ist es vorteilhaft bereits bei der Erstellung eines Quellmetamodells diesen Umstand zu berücksichtigen. Beispielsweise sollte eine Möglichkeit zur Markierung des letzten Elementes einer verketten Liste von Objekten vorgesehen werden, falls auf dieses Element im Rahmen einer Modelltransformation zugegriffen werden muss.

- Idealer Weise sollte nur eine Zielobjektvariable Attributwerte in Nicht-Primärschlüsselattribute von Objekten desselben Typs schreiben. Diese Maßnahme erleichtert es, die die Anwendbarkeit eines Regelwerks sicherzustellen.

- Alle nötigen Multiplizitätsuntergrenzen sollten nach Möglichkeit in jeder Zielmodellvariabeln bereits eingehalten werden, um die Untergrenzenkonformität des Regelwerks sicherstellen zu können.
- Zur Sicherstellung der Obergrenzenkonformität eines Regelwerks sollten Objektvariablenassoziationen aus verschiedenen Regeln als redundant identifizierbar sein. Hierzu kann in einer Regel die „Infrastruktur“ aus notwendigen Strukturen kopiert werden, während weitere Regeln Sonderfälle, also Erweiterungen dieser Kernfragmente abbilden.
- Zielobjektvariablen gleichen Typs, die disjunkte Mengen von Objekten erzeugen, sollten mit unähnlichen Identifikatoren gekennzeichnet werden oder aber von verschiedenen Arten von Quellobjektvariablen abhänigig sein. Dies erleichtert den Nachweis der Obergrenzenkonformität und der Konfliktfreiheit der Objektvariablen.

4.3 Bijektivität

Die dritte bedeutende Eigenschaft von Regeln und Regelwerken ist ihre Bijektivität. Ein Regelwerk ist bijektiv bezüglich eines Quellmetamodells, falls die umgekehrte Anwendung aller Regeln mit diesem Quellmetamodell aus einem erzeugten Zielmodell wieder genau das Quellmodell mit dem entsprechendem Quellmetamodell erzeugt. Im Folgenden wird formal festgehalten, wann Regelwerke bijektiv bezüglich eines Quellmetamodells sind und eine vergleichsweise einfache Verifikationstechnik für den Nachweis der Bijektivät von Regelwerken vorgestellt.

Bei bijektiven Abbildungen zwischen Modellen wird zwischen strenger und „normaler“ Bijektivität unterschieden Die Abbildung eines Modells und die anschließende Umkehrabbildung des erzeugten Modells liefert bei streng bijektiven Abbildung wieder exakt das Ursprungsmodell. Bei „normal“ bijektiven Abbildungen liefert diese Transformation ein zum Ursprungsmodell isomorphes Modell. Dieses gleicht dem Ursprungsmodell in seiner Struktur und den Attributbelegungen, jedoch können sich die Identifikatorwerte im so erzeugten Modell von denen im Ursprungsmodell unterscheiden. Diese Form der Bijektivität spielt vor allem bei der Realisierung bijektivier Abbildungen von Modellen in einem technischen Format (wie z.B. Java-Objektstrukturen) eine Rolle, da hier die Identität erzeugter Objekte oftmals nicht explizit bestimmt werden kann. Die nachfolgende Definition legt den Begriff der Isomporhie zwischen Modellfragmenten formal fest.

Definition 4.3.1 (Isomorphe Modellfragmente ($mf_0 \sim mf_1$))
Zwei Modellfragmente $mf_0, mf_1 \in \mathbb{MF}$ heißen *isomorph,* falls die im Folgenden definierte Relation $mf_0 \sim mf_1$ gilt:

$\sim : \mathbb{MF} \times \mathbb{MF} \to \mathbb{B}$
$mf_0 \sim mf_1 :\Leftrightarrow$

(i) $mf_0|_{mm} \cong mf_1|_{mm} \wedge$

(ii) es existiert eine bijektive Abbildung $map_o : mf_0|_{OB_{mm_0}} \to mf_1|_{OB_{mm_1}}$ mit

$$\forall o_0 \in mf_0|_{OB_{mm_0}} : map_o(o_0) = o_1 \wedge o_0|_{ot} = o_1|_{ot} \wedge o_0|_V = o_1|_V \qquad \wedge$$

(iii) es existiert eine bijektive Abbildung $map_a : mf_0|_{OA} \to mf_1|_{OA}$ mit

$$map_a(a_0) = a_1, \text{ mit } \forall a_0, a_1 : a_0 = (o_0, o_1, oat, card) \Leftrightarrow a_1 = (map_o(o_0), map_o(o_1), oat, card)$$

❍

Dementsprechend dürfen sich in zwei isomorphen Modellfragmenten lediglich die Identifikatoren von Objekten unterschieden. Die Struktur der beiden Fragmente und die Werte der Attribute müssen jedoch gleich sein.

Wie in Abschnitt 2.2.1 bereits festgehalten wurde existieren drei Möglichkeiten für die Realisierung bijektiver Abbildungen: zu einer Transformationsspezifikation lässt sich eine Umkehrabbildung generieren, jede Transformationsspezifikation ist generell bijektiv oder aber es existiert eine Klasse von Transformationen, für welche die Bijektivität nachgewiesen werden kann. Der hier vorgestellte Ansatz entspricht dem letzten Fall. Es wird versucht für ein BOTL-Regelwerk nachzuweisen, dass seine umgekehrte Anwendung eine Umkehrabbildung des ursprünglichen Regelwerks darstellt.

Zunächst wird nun der Begriff der *strengen Bijektivität* formal gefasst:

Definition 4.3.2 (Streng bijektive Regelwerke)
Ein Regelwerk $R \in \mathbb{RW}$ heißt *streng bijektiv* bezüglich des Quellmetamodells $mm_0 \in MM_R^0$, genau dann wenn gilt:

$$\forall M \in \mathbb{M}_R^0 : transform(transform(M,R), R_{mm_0}^{-1}) = m \in M : m|_{mm} = mm_0 \qquad \text{❍}$$

Analog hierzu werden *bijektive Regelwerke* definiert, deren Rücktransformation jeweils nur ein zum Ursprungsmodell isomorphes Modell liefern muss:

Definition 4.3.3 (Bijektive Regelwerke)
Ein Regelwerk $R \in \mathbb{RW}$ heißt *bijektiv* bezüglich des Quellmetamodells $mm_0 MM_R^0$, genau dann wenn gilt:

$$\forall M \in \mathbb{M}_R^0 : transform(transform(M,R), R_{mm_0}^{-1}) \sim m \in M : m|_{mm} = mm_0 \qquad \text{❍}$$

Die Abbildung *srcMatches* liefert zu einem Regelwerk und einer Menge von Quellmodellen sämtliche Matches von Quellmodellvariablen in den Quellmodellen, die im Verlauf der Anwendung des Regelwerks auftreten. Die Definition wird im weiteren Verlauf für die Definition sogenannter „Match-bijektiver"Regelwerke benötigt.

Definition 4.3.4 (Quell-Matches)
Sei $R \in \mathbb{RW}$ ein Regelwerk und $M \in \mathbb{M}_R^0$ eine Menge von Quellmodellen. Die Menge aller *Quell-Matches* von R in M ist definiert als

$$srcMatches : \mathbb{RW} \times \mathscr{P}\mathbb{M}) \rightarrow \mathscr{P}(\mathbb{MFM})$$
$$srcMatches(M,R) := \bigcup_{r \in R} MFM(srcModel(M,r), r|_{mv_0}) \qquad \text{❍}$$

Es wird nun die Eigenschaft eines Regelwerks definiert, dass bei der Anwendung des Umkehrregelwerks bezüglich eines Quellmetamodells die gefundenen und erzeugten Modellfragmente im entsprechenden Quellmodell bzw. dem Zielmodell genau wieder aufeinander abgebildet werden.

Definition 4.3.5 (Streng match-bijektive Regelwerke)
Sei $R \in \mathbb{RW}$ ein Regelwerk, $M \in \mathbb{M}_R^0$, R heißt *streng match-bijektiv* bezüglich des Quellmetamodells mm_0, wenn gilt:

(i) R ist metamodellkonform.

(*ii*) $R^{-1}_{mm_0}$ ist metamodellkonform.

(*iii*) Sei $R' = \{r_i \in R : r_i|_{mv_0} = mm_0\}$ und es existiert eine bijektive Abbildung

$$map : srcMatches(M,R') \rightarrow srcMatches(transform(M,R),(R)^{-1}_{mm_0})$$

so dass gilt:
Sei $r \in R'$ die Regel, während deren Anwendung der Match mfm_i auftritt (d.h. $mfm_i \in MFM(srcModel(M,R'),r)$).

$$mft(mfm_i,r) = map(mfm_i)|_{mf} \tag{4.30}$$
$$mft(map(mfm_i),r^{-1}) = mfm_i|_{mf} \tag{4.31}$$

(*iv*) $\bigcup_{\mathfrak{m}_{mfm \in srcMatches(M,R')}} mfm|_{mf} = m \in M : m_{mm} = mm_0$ ❍

Die Abbildung *map* ordnet jedem Match im Quellmodell genau einen Match des Umkehrregelwerks im Zielmodell zu. (4.30) besagt, dass das jeweilige Quellmodellfragment in das durch *map* zugeordnete Zielmodellfragment transformiert wird. (4.31) legt fest, dass die jeweilige Umkehrregel dieses Zielmodellfragment wieder genau in das entsprechende Quellmodellfragment abbildet. Da *map* bijektiv ist existiert kein weiterer Match des Umkehrregelwerks im Zielmodell der sich keinem Match des ursprünglichen Regelwerks zuordnen lässt.

Aussage (*iv*) besagt schließlich, dass die Menge aller Quellmatches von R das gesamte Quellmodell m erfasst, d.h. alle Elemente dieses Quellmodells werden durch das Regelwerk erfasst.

Der Umstand, dass streng Match-bijektive Regelwerke streng bijektiv sind, ist bereits intuitiv erfassbar. Das nachfolgende Lemma hält diese Tatsache fest:

Lemma 4.3.1 (Streng match-bijektive Regelwerke sind streng bijektiv)
Sei $R \in \mathbb{RW}$ ein bezüglich eines Quellmetamodells $mm_0 \in MM^0_R$ streng match-bijektives Regelwerk. Dann gilt: R ist streng bijektiv bezüglich mm_0. (Beweis s. A.2.3, S. 307) ❑

Analog zu streng match-bijektiven Regelwerken werden nun match-bijektive Regelwerke eingeführt.

Definition 4.3.6 (Match-bijektive Regelwerke)
Sei $R \in \mathbb{RW}$ ein Regelwerk, $M \in \mathbb{M}^0_R$, R heißt *match-bijektiv* bezüglich des Quellmetamodells $mm_0 \in MM^0_R$, wenn gilt:

(*i*) R ist metamodellkonform.

(*ii*) $R^{-1}_{mm_0}$ ist metamodellkonform.

(*iii*) Sei $R' = \{r_i \in R : r_i|_{mv_0} = mm_0\}$ und es existiert eine bijektive Abbildung

$$map : srcMatches(M,R') \rightarrow srcMatches(transform(M,R),(R)^{-1}_{mm_0})$$

so dass gilt:
Sei $r \in R'$ die Regel, während deren Anwendung der Match mfm_i auftritt (d.h. $mfm_i \in MFM(srcModel(M,R'),r)$).

$$mft(mfm_i,r) \sim map(mfm_i)|_{mf} \tag{4.32}$$
$$mft(map(mfm_i),r^{-1}) \sim mfm_i|_{mf} \tag{4.33}$$

$$(iv) \quad \bigcup_{\mathfrak{m} \atop mfm \in srcMatches(M,R')} mfm|_{mf} = m \in M : m_{mm} = mm_0 \qquad \text{❍}$$

Selbstverständlich gilt auch für Match-bijektive Regelwerke, dass diese bijektiv sind. Dies ist die Aussage des folgenden Lemmas.

Satz 4.3.1 (Match-bijektive Regelwerke sind bijektiv)
Sei $R \in \mathbb{RW}$ ein bezüglich eines Quellmetamodells $mm_0 \in MM_R^0$ match-bijektives Regelwerk. Dann gilt: R ist streng bijektiv bezüglich mm_0. ❑

Der Beweis verläuft vollkommen analog zum Beweis von Satz 4.3.1.

Eine Möglichkeit um die strenge Bijektivität eines Regelwerks nachzuweisen besteht demnach darin zu zeigen, dass es sich bei dem Regelwerk um ein streng Match-bijektives Regelwerk handelt. Der nachfolgende Satz macht sich diesen Umstand zunutze, indem er eine Reihe von Kriterien festlegt, anhand derer sich die Bijektivität eines Regelwerks nachweisen lässt.

Satz 4.3.2 (Strenge Bijektivität von Regelwerken)
Ein Regelwerk $R \in \mathbb{RW}$ ist streng bijektiv bezüglich eines Quellmetamodells $mm_0 \in MM_R^0$, falls gilt:

1. R und $R_{mm_0}^{-1}$ sind metamodellkonform.
2. Jeder Assoziationstyp kommt nur einmal innerhalb des Regelwerks vor.
3. Für jede Klasse die in einem der Quellmetamodelle oder im Zielmetamodell vorkommt existiert innerhalb der Regelwerks R höchstens eine Objektvariable dieses Typs.
4. Alle Modellvariablen in Regeln aus R sind zusammenhängend.
5. Für jede Regel aus $\{r \in R : r|_{mv_0}|_{mm} = mm_0\}$ gilt:
   a) Kein Identifikator in r hat den Wert $\diamondsuit$
   b) $\neg constrained(r) \wedge \neg constrained(r^{-1})$
   c) Alle Objekte und Assoziationen werden transformiert.
6. Die Struktur jedes Modells $m \in \mathbb{M}_{mm_0}$ lässt sich aus Instanzen der Quellmodellvariablen der Regeln in $\{r \in R : r|_{mv_0}|_{mm} = mm_0\}$ zusammensetzen.

(Beweis s. A.2.3, S. 308) ❑

Die nicht strenge Bijektivität von Regelwerken kann analog nachgewiesen werden, es wird an dieser Stelle jedoch auf die Angabe eines eigenen Satzes hierfür verzichtet.

Die hier vorgestellte allgemeine Verifikationstechnik für die strenge Bijektivität erlaubt den Nachweis dieser Eigenschaft lediglich für eine kleine Klasse an Regelwerken. Insbesondere die Einschränkungen, dass jeder Typ einer Objektvariablen und einer Objektvariablenassoziation innerhalb des Regelwerks nur einmal auftreten darf erweisen sich in der Praxis oftmals als zu restriktiv. Alternativ kann die strenge Bijektivität eines Regelwerks durch auf den Einzelfall angepasste Beweistechniken für die strenge Match-Bijektivität des Regelwerks nachgewiesen werden. Ein Beispiel hierfür findet sich in Abschnitt 5.2.3.

4.4 Metamodellkonformität von Rewrite-Regelwerken

Die folgende Definition führt den Begriff Metamodellkonformität für Rewrite-Regelwerke ein. Ein Rewrite Regelwerk wir als metamodellkonform bezeichnet, falls jede Anwendungsreihenfolge der Regeln für beliebige aber gültige Quellmodelle immer zu einem metamodellkonformen Zielmodell führt.

Definition 4.4.1 (Metamodellkonformes Rewrite-Regelwerk)
Ein Rewrite-Regelwerk $RW \in \mathbb{RRW}$ heißt *metamodellkonform*, genau dann wenn das im Folgenden definierte Prädikat $mmConform(RW)$ gilt:

$$
\begin{aligned}
&mmConform : \mathbb{RRW} \to \mathbb{B} \\
&mmConform(RW) :\Leftrightarrow \\
&\quad \forall M \cup \{m\} \in \mathbb{M}^0_{RW}, (t_0,\ldots,t_n) \in (\mathbb{N}^+_0 \times \cdots \times \mathbb{N}^+_0) \text{ mit } m|_{mm} = mm^1_{RW} \wedge C1(m) : \\
&\qquad rwTransform(M \cup \{m\}, RW, (t_0,\ldots,t_n)) \in \mathbb{M}^1_{RW}
\end{aligned}
$$ ❍

Im Folgenden wird durch eine Reihe von Lemmata belegt, dass alle denkbaren C1-Identitäts-Regelwerke immer metamodellkonform sind. Die einzelnen Aussagen werden im Weiteren verwendet, um die Metamodellkonformität von Rewrite-Regelwerken leichter nachweisen zu können. Zunächst wird in Lemma 4.4.1 die Aussage festgehalten, dass sämtlichen Objektvariable eines C1-Identitäts-Regelwerks konfliktfrei sind.

Lemma 4.4.1 (Zielobjektvariablen in C1-Identitätsregelwerken sind konfliktfrei)
Sei $mm_0 \in \mathbb{MM}$ beliebig. Dann gilt:

$$\forall ov_i, ov_j \in R^{id}_{mm_0}|_{mv_1}|_{OVB} : conflictFree(ov_i, ov_j)$$ (Beweis s. A.2.4, S. 309) ❑

In Lemma 4.4.2 wird allgemein gezeigt, dass jede Regel eines beliebigen C1-Identitäts-Regelwerks anwendbar ist.

Lemma 4.4.2 (Regeln in C1-Identitätsregelwerken sind anwendbar)
Sei $mm_0 \in \mathbb{MM}$ beliebig. Dann gilt:

$$\forall r \in R^{id}_{mm_0} : r \text{ ist anwendbar.}$$ (Beweis s. A.2.4, S. 310) ❑

Mit Hilfe des vorangegangen Lemmas wird in Lemma 4.4.2 nachgewiesen, dass beliebige C1-Identitäts-Regelwerke immer anwendbar sind.

Lemma 4.4.3 (C1-Identitätsregelwerke sind anwendbar)
Sei $mm_0 \in \mathbb{MM}$ beliebig. Dann gilt: $R^{id}_{mm_0}$ ist anwendbar. (Beweis s. A.2.4, S. 312) ❑

Für den Nachweis der Metamodellkonformität von C1-Identitäts-Regelwerken muss nun gemäß Satz 4.2.7 gezeigt werden, dass solche Regelwerke immer untergrenzen- und obergrenzenkonform sind. Diese Aussagen werden in den Lemmata 4.4.4 und 4.4.6 festgehalten.

Lemma 4.4.4 (C1-Identitätsregelwerke sind obergrenzenkonform)
Sei $mm_0 \in \mathbb{MM}$ beliebig. Dann gilt: $R^{id}_{mm_0}$ ist obergrenzenkonform. (Beweis s. A.2.4, S. 312) ❑

Lemma 4.4.5 (Regeln aus C1-Identitätsregelwerken sind untergrenzenkonform)
Sei $mm_0 \in \mathbb{MM}$ beliebig. Dann gilt:

$$\forall r \in R^{id'}_{mm_0} : lbConfrom(r)$$ (Beweis s. A.2.4, S. 313) ❑

Lemma 4.4.6 (C1-Identitätsregelwerke sind untergrenzenkonform)
Sei $mm_0 \in \mathbb{MM}$ beliebig. Dann gilt: $R^{id}_{mm_0}$ ist untergrenzenkonform. (Beweis s. A.2.4, S. 313) ❑

Für den Nachweis der Metamodellkonformität eines Rewrite-Regelwerks gilt es zu verifizieren, dass eine Menge von Rewrite-Regeln zusammen mit dem C1-Identitäts-Regelwerk metamodellkonform ist. Die Metamodellkonformität eines C1-Identitäts-Regelwerk ist hierfür zwar keine Voraussetzung, dennoch soll an dieser Stelle festgehalten werden, dass auch jedes beliebige C1-Identitäts-Regelwerk für sich metamodellkonform ist.

Satz 4.4.1 (C1-Identitätsregelwerke sind metamodellkonform)
Sei $mm_0 \in \mathbb{MM}$ beliebig. Dann gilt: $R^{id}_{mm_0}$ ist metamodellkonform. (Beweis s. A.2.4, S. 313) ❑

Der nachfolgende Satz bietet eine Verifikationstechnik um die Metamodellkonformität eines Rewrite-Regelwerks nachweisen zu können. Diese stützt sich auf die Metamodellkonformität gewöhnlicher BOTL-Regelwerke.

Satz 4.4.2 (Metamodellkonforme Rewrite-Regelwerke)
Ein Rewrite-Regelwerk $RW \in \mathbb{RRW}$ erzeugt ausschließlich metamodellkonforme Modelle falls gilt:

$$\forall i \in \mathbb{N}_0^+ \text{ mit } \exists r \in RW : tx(RW, r) = i : mmConform(R^{id}_{mm} \cup RW^i)$$

(Beweis s. A.2.4, S. 314) ❑

Der Nachweis der Metamodellkonformität eines Rewrite-Regelwerks RW mit $mm^1_{RW} = mm_n$ erfordert demnach den Nachweis der Metamodellkonformität für eine Menge von Regelwerken der Form $R^{id}_{mm_n} \cup RW^i$. Das folgende Lemma erleichtert diese Aufgabe, indem es eine Verifikationstechnik für die Anwendbarkeit solcher Regelwerke anbietet.

Lemma 4.4.7 (Anwendbarkeit von Rewrite-Regelwerken)
Sei $RW \in \mathbb{RRW}$ ein Rewrite-Regelwerk mit dem Zielmetamodell $mm_n := mm^1_{RW}$. Für ein beliebiges $i \in \mathbb{N}_0^+$ sei

$$RW^i := \{r \in RW : tx(RW, r) = i\}$$

Dann ist das Regelwerk $R^{id}_{mm_n} \cup RW^i$ anwendbar, falls gilt:

(i) $\forall r \in RW^i : r$ ist anwendbar

(ii) $\forall (ov_i, ov_j) \in (RW^i|_{mv_1}|_{OVB} \times R^{id}_{mm_n}|_{mv_1}|_{OVB} \cup RW^i|_{mv_1}|_{OVB}) :$
$conflictFree(ov_i, ov_j)$ (Beweis s. A.2.4, S. 314) ❑

Das Lemma erlaubt den Nachweis der Anwendbarkeit eines Rewrite-Regelwerks, falls die Anwendbarkeit der einzelnen Regeln bereits an anderer Stelle nachgewiesen wurde. Zudem muss die Anwendbarkeit der Regeln des Identitäts-Regelwerks nicht mehr im Einzelfall nachgewiesen werden. Ein Beispiel für die Verwendung dieses Lemmas findet sich in Abschnitt 5.4.

4.5 Semantik von Transformationsregeln

Während objektorientierte Modelle gut für eine werkzeuginterne Repräsentation von Systemen und ihrer Umgebung geeignet sind, erlauben es mathematisch fundierte Kalküle zur Modellierung von Software-Systemen, Aussagen über Eigenschaften der modellierten Systeme zu machen. Auch Anforderungen an ein System zur Erfüllung der Konsistenz und der Vollständigkeit können auf Basis mathematischer Kalküle in der Regel wesentlich schärfer gefasst werden als durch objektorientierte Metamodelle.

Aus diesem Grunde ist es notwendig, für objektorientierte Metamodelle eine Semantik zu definieren, in der festgelegt ist, wie eine Instanz eines Metamodells auf ein semantisches Modell auf der Grundlage eines gegebenen Kalküls zu interpretieren ist (siehe auch Abschnitt 2.1, S. 23 ff).

Ansätze zur Transformation von UML-Diagrammen in mathematische Spezifikationen finden sich z.B. in [VGP01], eine fundierte Transformationssprache hierzu existiert jedoch bisher noch nicht. In [Sae02] werden Graphgrammatiken verwendet um UML-Spezifikationen mit einer Semantik (hier in Form von Z-Spezifikationen) zu unterlegen.

Ist eine eine Abbildung objektorientierter Modelle auf semantische Modelle möglich, so ist es weiterhin wünschenswert auch Aussagen über die Auswirkung einer Transformation objektorientierter Modelle auf Basis des jeweils verwendeten Kalküls zu machen. Im Rahmen dieses Abschnitts wird skizziert, wie sich eine Semantik für objektorientierte Modelle definieren lässt und wie sich Aussagen über die Auswirkung von BOTL-Transformationen auf Basis des jeweils verwendeten Kalküls machen lassen.

Als begleitendes Beispiel werden im folgenden nichtdeterministische endliche Automaten als semantisches Kalkül verwendet. Ein Automat M ist (gemäß [Sch93]) als ein 5-Tupel

$$M = (Z, \Sigma, \sigma, S, E)$$

definiert, wobei gilt:

- Z ist die endliche Menge der *Zustände* des Automaten
- Σ ist das endliche *Eingabealphabet* des Automaten mit $Z \cup \Sigma = \emptyset$
- $\sigma : Z \times \Sigma \rightarrow \mathscr{P}(Z)$ ist die *Überführungsfunktion*
- $S \subseteq Z$ ist die Menge der *Startzustände*
- $E \subseteq Z$ ist die Menge der *Endzustände*

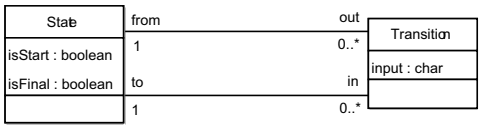


Abbildung 4.35: Das Metamodell mm_A

Abbildung 4.35 stellt das Metamodell mm_A für die Repräsentation von Automaten in Form objektorientierter Modelle dar. Durch die Attribute isStart und isFinal der Klasse State wird angegeben, ob ein Zustand Teil der Menge der Start- bzw. Endzustände eines Automaten ist. Die Klasse Transistion modelliert Zustandsübergänge von einem Zustand zum nächsten, wobei das Attribut input das jeweils von der Transition gelesene Zeichen des Eingabealphabetes enthält.

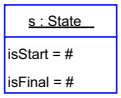
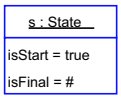
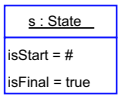
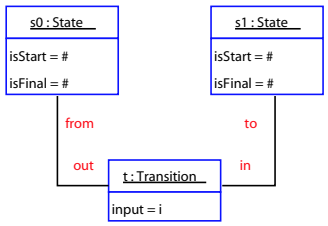
Modellvariable:	Semantik:
	$s \in Z$
	$s \in S$
	$s \in E$
	$i \in \Sigma \wedge$ $s1 \in \sigma(s0, i)$

Tabelle 4.4: Semantik für das Metamodell mm_A

Um für ein Metamodell eine Semantik anzugeben, gilt es seinen Instanzen eine Menge von Aussagen, welche auf einem Kalkül basieren zuzuordnen. In der Praxis erfolgt diese Zuordnung sehr oft anhand natürlichsprachlichen Textes. Eine präziesere Zuordnung ist möglich, indem anstelle von textuellen Beschreibungen Modellvariablen verwendet werden, die Teile der Struktur von objektorientierten Modellen beschreiben. Wird einer Modellvariablen eine Menge von Aussagen zugeordnet, so gelten diese Aussagen für alle Modellfragmenten des Modells in denen die Modellvariable matcht. Ein semantisches Modell eines objektorientierten Modells erhält man als die Konjunktion aller diese Aussagen für alle Zuordnungen von Aussagen zu Modellvariablen einer so definierten Semantik.

Die Semantik des Metamodells mm_A kann also definiert werden, indem Modellvariablen Mengen von Aussagen zugeordnete werden. Tabelle 4.4 stellt eine Definition der Semantik für das Metamodell mm_A auf Basis von nichtdeterministischen Automaten vor [1]. Damit eine gültige semantische Abbildung durch sie definiert ist, darf durch das Vorhandensein zusätzlicher Elemente im Objektmodell keine der Aussagen ungültig werden.

Zu einem Quellmodell m_0 kann auf diese Weise ein semantisches Modell aus einer Menge M_0 von Elementen mit Eigenschaften $P(M_0)$ angegeben werden. Dementsprechend lässt sich für jede Modellvariable einer Regel $r = (mv_0, mv_1)$ jeweils für die Menge M_1 von Elementen des semantischen Zielmodells m_1 eine Menge von semantischen Aussagen $P(M_1)$ formulieren. Generell können dann über das semantische Modell eines Zielmodells Aussagen der folgenden Form gemacht werden:

$$\forall M_0 \in m_0 \text{ mit } P(M_0) : M_1 \in m_1 mit P(M_1)$$

Die in Abbildung 4.36 dargestellte Regel r bildet Instanzen des Metamodell mm_A auf ein isomorphes Metamodell ab. Hierbei wird die Richtung sämtlicher Transitionen umgekehrt, Startzustände werden zu Endzuständen und umgekehrt. Auf Basis der in Tabelle 4.4 lässt sich dieser Abbildung für einen Quellautomaten $M = (Z, \Sigma, \sigma, S, E)$ formal die folgende Aussage über den erzeugten Zielauto-

[1] Der Einfachheit halber berücksichtigt die hier vorgestellte Definition lediglich Automaten ohne reflexive Transitionen. D.h. es muss gelten: $\forall s \in Z, i \in \Sigma : s \notin \sigma(s, i)$

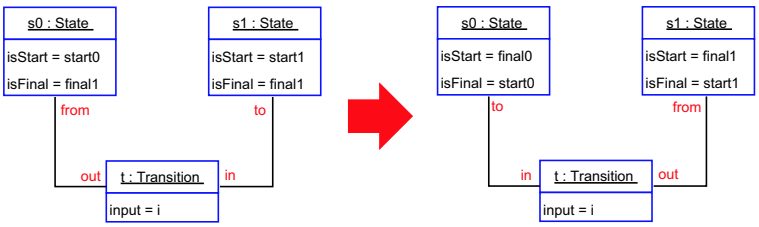


Abbildung 4.36: Die Regel r

maten $M' = (Z', \Sigma', \sigma', S', E')$ zuordnen:

$$\begin{aligned} &\forall\, s0, s1 \in Z, i \in \Sigma \text{ mit } start0 := (s0 \in S),\, final0 := (s0 \in E), \\ &\qquad\qquad start1 := (s1 \in S),\, final1 := (s1 \in E), s1 \in \sigma(s0, i) : \\ &s0, s1 \in Z', i \in \Sigma' \text{ mit } \; (s0 \in S') = final0, (s0 \in E') = start0, \\ &\qquad\qquad (s1 \in S') = final1, (s1 \in E') = start1, s0 \in \sigma'(s1, i) \end{aligned} \tag{4.34}$$

Für einen gegebenen, zusammenhängenden Ausgangsautomaten M lassen sich anhand dieser Aussage bereits Eigenschaften über den durch die Anwendung der Regel erzeugten Zielautomaten M' machen. Ist beispielsweise bekannt, dass der Automat M ein Wort w akzeptiert, so lässt sich mit Hilfe von Aussage (4.34) leicht nachweisen, dass der erzeugten Zielautomat M' das selbe Wort in umgekehrter w^{-1} Reihenfolge als Eingabe akzeptiert.

4.6 Einordnung der BOTL

Mit BOTL steht eine ausdrucksmächtige und intuitive Sprache zur Spezifikation von Transformationen objektorientierter Modelle zur Verfügung. BOTL lässt sich anhand des in Abschnitt 2.2.1 vorgestellten Klassifikationsschemas in folgender Weise charakterisieren:

Art der transformierten Modelle: BOTL transformiert objektorientierte Modelle (Objektmodelle). Die Struktur dieser Modelle wird durch ein Klassenmodell (wie z.B. das der MOF) festgelegt (siehe Abschnitt 3.1).

Kardinalität der Ein-/Ausgabemodelle: BOTL unterstützt sowohl n:1 Transformationen (siehe Abschnitt 3.3), als auch Rewrite-Transformationen auf einem Modell (siehe Abschnitt 3.6). Durch die Kombination mehrerer n:1-Transformationen lassen sich auch n:m Transformationen abbilden.

Verwendetes Sprachparadigma: BOTL verwendet eine regelbasierte, deklarative Sprache zur Spezifikation von Modelltransformationen. Auch für die Manipulation primitiver Typen erfolgt deklariv, indem der Zusammenhang zwischen Werten im Zielmodell und in den Quellmodellen durch arithmetische Ausdrücke spezifiziert wird. (siehe Abschnitt 3.3 und 3.5).

Verarbeitung primitiver Typen: Primitive Typen können mit beliebigen mathematischen Operationen oder String-Operationen (wie z.B. die Konkatenation) modifiziert werden (siehe Abschnitt 3.3).

Bijektivität: BOTL ermöglicht den Nachweis, dass Abbildungen bijektiv sind (siehe Abschnitt 4.3).

Rückverfolgbarkeit: Die Rückverfolgbarkeit von Elementen des Zielmodells kann bei entsprechender Gestaltung der Transformationsregeln gewährleistet werden.

Kopplung der Modelle: Quell- und Zielmodelle sind nicht gekoppelt. Konzeptuelle Modelle, wie sie im Rahmen dieser Arbeit betrachtet werden, werden jedoch ausschließlich durch die Verfeinerung abstrakterer konzeptueller Modelle und und der Integration zusätzlicher Artefakte gebildet. Änderungen in einer der Quellen können somit durch eine erneute Transformation aller Quellen in eine neue Instanz des Zielmodells berücksichtigt werden.

Korrektheit von Spezifikationen: Für eine BOTL-Spezifikationen lässt sich anhand der in Abschnitt 4.1 vorgestellten Verifikationstechniken formal nachweisen, dass diese Ausführbar ist. Auch für die den Nachweis, dass eine BOTL-Spezifikation nur metamodellkonforme Modelle erzeugt, existieren Verifikationstechniken (siehe Abschnitt und 4.2).

Fehlerverhalten: Ist eine deterministische Abbildung nicht möglich so wird die Transformation abgebrochen. Deterministische Abbildungen die zu einem nicht metamodellkonformen Ergebnis führen sind jedoch möglich (siehe Abschnitt 3.5).

Mächtigkeit: Die Mächtigkeit der BOTL entspricht der Muster-basierter Ansätze. Zu transformierende Fragmente in Quellmodellen lassen sich anhand ihrer Struktur, der Werte ihrer Identifikatoren und Attribute, sowie möglichen Relationen zwischen diesen Werten auswählen. Die erzeugten Fragmente können beliebige Strukturen aufweisen und sämtliche im Quellfragment enthaltenen Werte zur Berechnung ihrer Attributwerte und Identifikatoren verwenden. Eine Regelanwendung in Abhängigkeit von der Nicht-Existenz eines Elements in einem Quellmodell ist jedoch nicht möglich (siehe Abschnitt 3.5).

Anwenderfreundlichkeit:

- **Lesbarkeit:** BOTL verfügt über eine graphische Notation die sich an die UML anlehnt. Daher sind BOTL-Spezifikationen auch für nicht mit dem Formalismus vertraute Personen gut lesbar (siehe Abschnitt 3.4).
- **Verständlichkeit:** BOTL-Regeln sind sehr intuitiv zu interpretieren („Suche ein Muster wie links angegeben und erzeuge im Zielmodell ein Fragment wie auf der rechten Seite dargestellt.“). Somit ist die Auswirkung der Anwendung einer Regel für den Benutzer direkt ersichtlich.
- **Abstraktion:** Derzeit ist eine Abstraktion von Regelwerken lediglich durch das Weglassen von Regeln und die Verwendung parametrisierter Regeln möglich. Durch eine geeignete Organisation der Regeln eines Regelwerks ist es jedoch möglich dies so zu gruppieren, dass verschiedene Mengen Regeln jeweils unterschiedliche Aspekte einer Transformation abdecken.
- **Komponierbarkeit:** BOTL-Regelwerke können durch die Vereinigungsoperation komponiert werden. Der Nachweis der Anwendbarkeit und Metamodellkonformität eines so komponierten Regelwerks muss jedoch neu erbracht werden (siehe auch Satz 3.5.6 zur Komposition von Regelwerken).

Ausführbarkeit: Metamodellkonforme Transformationen sind automatisiert ausführbar. Dies wurde zum einen innerhalb von Kapitel 4 formal bewiesen, zum anderen existiert mit dem in Kapitel 6 eine Referenzimplementierung, die es erlaubt, Modelle automatisiert zu transformieren.

Effizienz: Die Effizienz der Transformationen entspricht der aller Ansätze die eine Mustersuche in Graphen realisieren müssen.

Realisierbarkeit: Der Nachweis der Realisierbarkeit das Ansatzes wurde anhand des in Kapitel 6 vorgestellten Werkzeuges erbracht.

Erweiterbarkeit: Generell ist der Einsatz angepasster Notationen problemlos möglich. Beispielsweise lassen sich Elemente der konkreten Syntax von Aktivitätsdiagrammen in Regeln darstellen, wobei Textfelder in den Diagrammen mit Termen beschriftet werden. Diese können dann entsprechend ihrer Darstellung in der abstrakten Syntax von BOTL interpretiert werden. Dieser Ansatz ist jedoch nicht für alle denkbaren Notationen gleich intuitiv, man bedenke hierbei z.B. Erfassung von Attributen in Klassendiagrammen zur Transformation.

BOTL erfüllt demzufolge die in den Abschnitten 2.2.2, 2.2.3 und 2.2.4 angeführten Anforderungen an eine Transformationssprache, sofern die verwendeten Modelle objektorientierte Modelle sind. Der Einsatz von BOTL für die Realisierung eines modellbasierten Entwicklungsprozesses wird innerhalb von Kapitel 5 exemplarisch dargestellt.

5 Anwendung der BOTL in einem modellbasierten Entwicklungsprozess

Im Rahmen dieses Abschnitts wird anhand einer modellbasierten Entwicklungsmethodik demonstriert, wie sich die in Kapitel 2 vorgestellten Ansätze und Konzepte mit Hilfe der Modelltransformationssprache BOTL realisieren lassen. Da BOTL hinreichend formal definiert ist, um eine automatisierte Verifikation und Ausführung von Regelwerken zu ermöglichen, ist eine direkte Umsetzung der vorgestellten Lösungsansätze in eine Werkzeugunterstützung ohne weiteres möglich.

In Abschnitt 5.1 wird zunächst die KOGITO-Methodik für das Requirements-Engineering Web-basierter Anwendungen vorgestellt. Innerhalb von KOGITO werden konzeptuelle Modelle auf verschiedenen Abstraktionsebenen durch automatisierte Verfeinerungsschritte erzeugt und schrittweise erweitert. Abschnitt 5.2 stellt eine solche Verfeinerungsabbildung zwischen konzeptuellen Modellen, wie sie in der KOGITO-Methodik verwendet werden, vor. Innerhalb von Abschnitt 5.3 wird gezeigt, wie sich unterschiedliche, im Verlauf des Entwicklungsprozesses erstellte Artefakte in ein konzeptuelles KOGITO-Modell integrieren lassen. Eine Diskussion dieser Integration von Artefakten in ein konzeptuelles KOGITO-Modell findet sich auch in [MS03d]. Die Dokumentation des Vorgehens durch Prozessmuster wird schließlich in Abschnitt 5.4 thematisiert.

5.1 Die KOGITO-Methodik

Innerhalb des BMBF-Projektes KOGITO [kog04, MS03b] wird derzeit eine Methodik für das Requirements Engineering Web-basierter Anwendungen entwickelt. Ziel des Projektes ist es, ein durchgängiges, werkzeugunterstütztes Vorgehen zu schaffen, welches Methoden und Techniken des Knowledge Management für das Requirements Engineering nutzt. Im Verlauf dieses Abschnitts werden die wesentlichen Modelle und Artefakte der Methodik, soweit sie für die exemplarische Darstellung der hier vorgestellten Techniken für modellbasierte Entwicklungen relevant sind, skizziert.

5.1.1 Umfeld und Verwandte Ansätze

Im Folgenden wird zunächst ein Überblick über das Umfeld der KOGITO-Methodik gewährt. Weiterhin werden zwei verwandte Ansätze zur modellbasierten Entwicklung Web-basierter Anwendungen vorgestellt.

Umfeld

Die Phase des Requirements Engineering gehört mit zu den ersten Aktivitäten im Verlauf eines Entwicklungsprojektes [Balzert,Polmann]. Ihr Ziel ist es die funktionalen und nicht-funktionalen Anforderungen an ein System vollständig und konsistent zu dokumentieren. Dies umfasst zunächst die Analyse der Anwendungsdomäne, sowie die Analyse und Spezifikation des Umfeldes der Anwendung aus fachlicher Sicht. Schließlich erfolgt die Spezifikation der fachlichen Anforderungen an das Sys-

tem und die Abbildung dieser Anforderungen auf technische Anforderungen an das zu erstellenden Software-System.

Während in den nachfolgenden Entwicklungsschritten der Einsatz von Modellen mit einer formalen Syntax zur Beschreibung der zu erstellenden Software mittlerweile zur Selbstverständlichkeit geworden ist, ist ihr Einsatz in den frühen Phasen des Requirements Engineering eher die Ausnahme. In der Regel werden Anforderungen in Form von mehr oder weniger strukturiertem Text dokumentiert. Existierende Werkzeuge zur Dokumentation und Verwaltung von Anforderungen wie RequisitePro [Cor04b] oder DOORS [Doo04] bieten dementsprechend lediglich die Möglichkeit Textfragmente zu verwalten und über einen Link-Mechanismus in Beziehung zueinander zu setzen.

Ein konzeptuelles Modell, welches die Konzepte und Zusammenhänge der Anwendungsdomäne und des zu erstellenden Systems ausdrückt, ist weder in den gängigen Werkzeugen, noch in den gängigen Methoden zum Requirements Engineering vorgesehen. Dies führt dazu, dass Anforderungen oft unpräzise formuliert sind. Eine automatische Verifikation der Widerspruchsfreiheit und ggf. Vollständigkeit der Anforderungen ist wegen der fehlenden Modellbildung nicht möglich. Auch eine systematische Abbildung von Anforderungen auf Elemente der Systemarchitektur in späteren Phasen der Entwicklung ist wegen des informellen Charakters von Anforderungsspezifikationen nicht oder nur manuell möglich. Durch das Fehlen domänenspezifischer Metamodelle als Grundlage eines Vorgehens zum Requirements Engineering ist es weiterhin nicht möglich einem Anwender gezielte Unterstützung bei der Ausarbeitung und Strukturierung von Anforderungen zu bieten.

Der Hauptgrund für das Fehlen geeigneter (Meta-) Modelle für das Requirements Engineering ist, dass die für die Modellierung des Anwendungsumfeldes nötigen Konzepte in der Regel domänenspezifisch sind. Die Anzahl der zur Spezifikation von Software-Systemen benötigten Konzepte ist vergleichsweise klein, in der Regel umfassen Metamodelle für die Modellierung von Software-Systemen eine überschaubare Menge von Entitätstypen wie Komponente, Konnektor, Datentyp, etc. Soll zusätzlich das fachliche Umfeld einer Anwendung modelliert werden, wie dies im Rahmen des Requirements Engineering erforderlich ist, so müssen die Modelle um domänenspezifische Konzepte erweitert werden. Folglich werden für unterschiedliche Anwendungsfelder verschiedene konzeptuelle Metamodelle benötigt. So benötigen beispielsweise Modelle zur Spezifikation von Unternehmensstrukturen und Geschäftsprozessen andere Ausdrucksmittel, als solche zur Modellierung von Realzeitkomponenten in der Bordelektronik von Flugzeugen. Die Verwendung unterschiedlicher Technologien in unterschiedlichen Anwendungsbereichen verstärkt zusätzlich die Notwendigkeit jeweils angepasste konzeptuelle Metamodelle für das Requirements Engineering zu verwenden.

Im Umfeld der Entwicklung Web-basierter Anwendungen existieren derzeit zwei Ansätze, welche auch die frühen Phasen der Entwicklung berücksichtigen und beide auf der UML basieren. Da diese Ansätze auch Einfluss auf die Gestaltung der KOGITO-Methodik hatten, werden sie im Folgenden kurz vorgestellt.

UML Profile for Enterprise Distributed Object Computing Specification (EDOC)

Die „UML Profile for Enterprise Distributed Object Computing Specification“ (EDOC) ist ein UML -Profil der OMG für die Modellierung verteilter, betrieblicher Anwendungen. EDOC umfasst ein auf der UML 1.4 basierendes Framework, welches zum Ziel hat, die einheitliche Spezifikation und Entwicklung verteilter Geschäftsanwendungen zu erleichtern. Innerhalb des in EDOC definierten Entwicklungsansatzes werden Systemspezifikationen in fünf Viewpoints gemäß dem "Reference Model of Open Distributed Processing"(RM-ODP) [ISO95] untergliedert.

EDOC sieht weiterhin verschiedene Abstraktionsgrade für die Modellierung von B2B und B2C Anwendungen vor. So umfasst EDOC mit der Enterprise Collaboration Architecture (ECA) ein UML-

Profil für plattformunabhängige Modelle (PIMs) gemäß der MDA. Die ECA selbst besteht aus fünf Teilprofilen:

- Die *Component Collaboration Architecture* (CCA) dient der Spezifikation von Struktur und Verhalten von Systemkomponenten. Die wichtigsten Modellelemente sind Komponenten, Ports, Protokolle und Dokumente. Zur Modellierung können alternativ UML-Klassendiagramme mit Stereotypen oder eine eigene, EDOC-spezifische Notation verwendet werden. Die Komposition von Komponenten erfolgt ebenfalls entweder mittels UML-Kollaborationsdiagrammen oder unter Verwendung einer eigenen CCA-Notation. Choreographien bezeichnen innerhalb der CCA den Ablauf der Kommunikation zwischen Komponenten. Sie werden mit Hilfe von UML-Aktivitätsdiagrammen modelliert.

- Die innerhalb der Anwendungsdomäne verwendeten Konzepte werden innerhalb von EDOC durch sogenannte Entitäten repräsentiert. Das *Entities Profile* ermöglicht die Spezifikation von Entitäten und deren Beziehungen untereinander. Als Notation werden hierzu UML-Klassendiagramme mit EDOC-spezifischen Stereotypen verwendet

- Das *Event Profile* erlaubt es, auch ereignisgesteuerte Anwendungssysteme und Geschäftsprozesse zu modellieren. Als Notation wird eine eigene EDOC-Notation verwendet, die sich jedoch stark an Konstrukte der UML anlehnt.

- Das *Business Process Profile* dient der Spezifikation von Geschäftsprozessen und stellt eine Verfeinerung des CCA dar.

- Das *Relationships Profile* definiert generische Relationen, welche eine strengere Semantik als die Standardrelationen der UML 1.4 haben können.

Die ECA erlaubt lediglich eine abstrakte Modellierung von kollaborierenden Anwendungen. Die Erstellung einer detaillierten technischen Spezifikation, der Schnittstellen und des Verhaltens von Anwendungen ist nicht mit Mitteln der ECA möglich. Hierzu enthält EDOC technologiespezifische Metamodelle für verschiedene Technologien. Die einzelnen technologiespezifischen Profile erlauben die Abbildung von ECA-Modellen auf plattformspezifische Modelle (PSMs) gemäß der MDA. Im Idealfall ist es somit möglich direkt Code aus plattformunabhängigen EDOC-Modellen für unterschiedliche Technologien zu generieren. Derzeit existieren technologiespezifische Profile für EJB-Anwendungen [Mic04] und für das Flow Composition Model (FCM), welches innerhalb der EDOC-Spezifikation eingeführt wird. Weitere Abbildungen auf bestehende Technologien sind beispielsweise für CORBA Components [OMG02a] und ebXML [eBPPT01] angekündigt.

Mit der EDOC steht somit eine Modellierungssprache für Web-basierte Anwendungen zur Verfügung, wie sie auch innerhalb von KOGITO benötigt wird. Allerdings basiert die Sprache vollständig auf dem UML-Metamodell, d.h. sie erweitert das UML-Metamodell durch zusätzliche Stereotypen und Tags. Dies führt dazu, dass das konzeptuelle Modell von EDOC ausgesprochen unübersichtlich wird. Nicht zuletzt wegen des großen Umfangs der EDOC-Spezifikation von deutlich über 400 Seiten bietet sich dieses Framework kaum für die Demonstration der im Rahmen dieser Arbeit vorgestellten Ansätze an.

UN/CEFACT Unified Modelling Methodology (UMM)

Die UN/CEFACT Unified Modelling Methodology (UMM) [UN/01] ist eine Methodik für die Entwicklung von Business-to-Business (B2B) Anwendungen. Sie basiert im Wesentlichen auf dem Uni-

fied Software Development Process [JBR99] und verwendet weitestgehend die Beschreibungstechniken der UML zur Modellierung. Das durch die UMM definierte Vorgehen umfasst vier Hauptphasen, welche von den frühen Aktivitäten der Software-Entwicklung über das Auffinden von Anforderungen bis hin zur Erstellung eines plattformunabhängigen Systemdesigns reichen. Die vier Entwicklungsphasen sind im Einzelnen [C.J00]:

Business Modelling: In dieser Phase wird ein Modell des Geschäftsumfeldes einer Anwendung erstellt. Ein solches Modell soll das gemeinsame Verständnis aller Beteiligten fördern und es ermöglichen Ansätze zur Verbesserung der bestehenden Geschäftsprozesse aufzufinden. Als Ergebnis dieser Phase werden erste abstrakte Anforderungen an das zu erstellende System gewonnen.

e-Business Requirements: Das Ziel dieser Entwicklungsphase ist es, die Anforderungen an das oder die Systeme zur Unterstützung des Geschäftsfeldes detaillierter auszuarbeiten. Hierzu wird zunächst ein einheitliches Glossar geschaffen. Die Anforderungen werden anhand von Nutzungsszenarien dokumentiert und relevante Geschäftsentitäten und Nutzerrollen identifiziert.

Analysis: Im Verlauf dieser Phase werden die zuvor aus Sicht der Anwender identifizierten Anforderungen analysiert und in eine für Software-Entwickler angemessene Anforderungsspezifikation des Systems umgewandelt. Es werden erste Systemkomponenten identifiziert und deren gegenseitige Nutzungsbeziehungen festgelegt.

Design: Innerhalb der Design-Phase werden Technologien für die Realisierung der in der Analysephase erarbeiteten Spezifikation ausgewählt. Es werden konkrete Nachrichtenformate für den Austausch von Daten festgelegt und ggf. bestehende Standards für die Kommunikation zwischen Systemen ausgewählt. Die im Rahmen der Analyse-Phase entstandene Anforderungsspezifikation wird dementsprechend weiter verfeinert und erweitert.

Die UMM sieht für jede dieser Phasen eine Reihe von Artefakttypen vor. Hierbei handelt es sich jeweils entweder um Diagrammtypen der UML oder verschiedene Arten von Formularen.

Die in den verschiedenen Artefakten enthaltene Information wird in ein gemeinsames konzeptuelles Modell integriert. Das Metamodell dieses gemeinsamen Modells ist ein Profil der UML. Dieses Profil ist in vier verschiedenen Sichten organisiert, die jeweils den einzelnen Entwicklungsphasen der UMM zuzuordnen sind.

Einige der Artefakttypen der UMM werden auch innerhalb der KOGITO-Methodik verwendet. Allerdings wird innerhalb von KOGITO ein eigenständiges konzeptuelles Metamodell verwendet,

5.1.2 Die konzeptuellen KOGITO-Metamodelle

Die KOGITO-Methodik basiert auf konzeptuellen Metamodellen verschiedenen Abstraktionsgrades, die speziell auf das Requirements Engineering von Web-basierten B2B-Anwendungen zugeschnitten sind. Dies ermöglicht den Einsatz angepasster Beschreibungstechniken, welche ebenfalls genau auf das Anwendungsgebiet abgestimmt sind und deren Informationsgehalt jeweils in ein konzeptuelles KOGITO-Modell abgebildet werden kann. Im Folgenden werden die konzeptuellen Metamodelle und die verwendeten Artefakttypen der KOGITO-Methodik kurz vorgestellt.

Alle in KOGITO verwendeten konzeptuelle Metamodelle sind Instanzen der MOF, die von der OMG zur Modellierung von Metamodellen vorgesehen ist (siehe Abschnitt 1.3.1. Im Gegensatz zu Ansätzen wie EDOC wird hier nicht das bestehende UML-Metamodell durch ein UML-Profil erweitert, da sich dieses bereits in seiner Ursprungsform als zu komplex und unübersichtlich erwiesen

hat um eine geeignete Grundlage zu bieten. Dies resultiert im Wesentlichen aus dem Umstand, dass sich im UML Metamodell sämtliche Beschreibungstechniken der UML wiederspiegeln. Elemente von UML-Diagrammen werden direkt auf Elemente des UML-Modells abgebildet. Diese Abbildung, und somit die Semantik der UML-Diagramme, ist lediglich textuell spezifiziert. Der in KOGITO verwendete Ansatz unterscheidet sich dadurch, dass Beschreibungstechniken von den konzeptuellen Modellen getrennt gehalten werden. Ihre Semantik erhalten die KOGITO-Beschreibungstechniken durch formal spezifizierte Abbildungsvorschriften. Dies erlaubt auch eine spätere Integration neuer oder die Veränderung bestehender Beschreibungstechniken, ohne dass eine Änderung im konzeptionellen Metamodell nötig wäre.

Dieser Ansatz erlaubt es, neben der Integration von unterschiedlichen Entwicklungsartefakten in ein konzeptuelles Modell, auch standardisierten Spezifikationen, wie sie im Bereich von B2B-Frameworks verbreitet sind, in das Modell zu integrieren. So ist möglich eine maschinenlesbare Spezifikationen wie z.B. BizTalk- [Cor00], ebXML- [eBPPT01], oder BPEL4WS-Spezifikationen [CGK+02] in ein konzeptuelles KOGITO-Modell zu überführen.

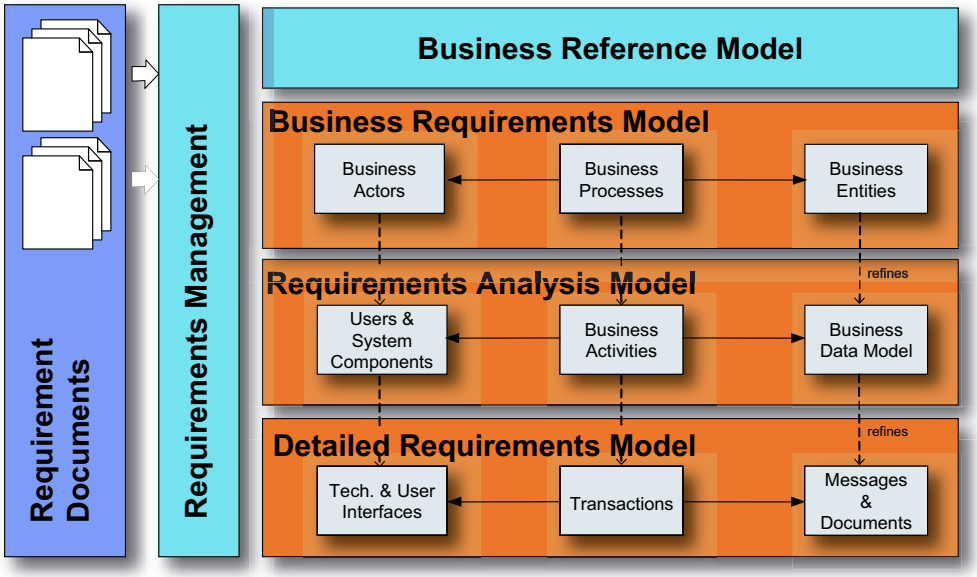


Abbildung 5.1: Die in KOGITO verwendeten konzeptuellen Metamodelle

Abbildung 5.1 gibt einen Überblick über die in KOGITO verwendeten konzeptuellen Metamodelle. Im Wesentlichen existieren innerhalb von KOGITO konzeptuelle Modelle auf drei Abstraktionsebenen. Die oberste, abstrakteste Schicht, das *Business Requirementes Modell* dient zur Modellierung der Anwendungsdomäne. Im *Requirements Analysis Modell* wird das System und Teile seiner Umgebung technologieunabhängig spezifiziert. Die konkreteste Ebene, das *Detailed Requirements Model*, ermöglicht bereits eine technologiespezifische Beschreibung des Systems. Die links abgebildete, vertikale Schicht stellt die *Requirements Management* Schicht dar, welche im Wesentlichen die Infrastruktur für die Verwaltung und Zuordnung von Artefakten in die konzeptuellen Modelle bildet. Zusätzlich existiert ein *Business Reference Modell*, das als Glossar während der Entwicklung dient.

Die in der Abbildung horizontal dargestellte Dreischichtung entspricht auch den drei Abstraktionsebenen wie sie auch von der MDA unverbindlich vorgeschlagen wird. So lassen sich die drei Modelle als Business Domain Model (BSM), Plattform Independent Model (PIM) und Plattform Specific Model (PSM) einer MDA Architektur auffassen. Da sich KOGITO jedoch auf die Phase des Requirements

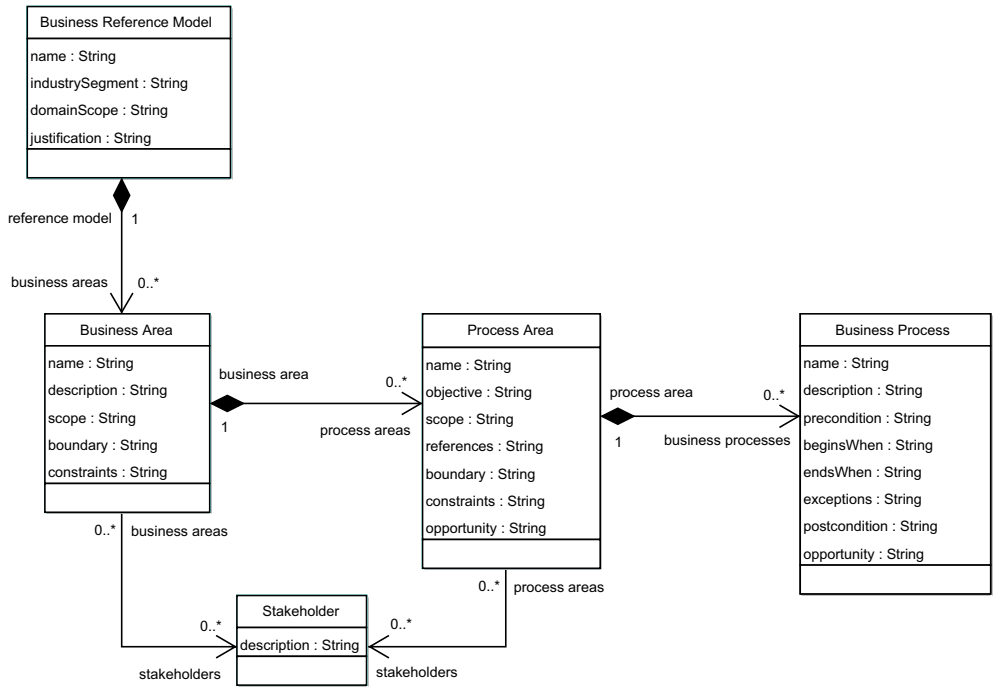


Abbildung 5.2: Das Business Reference Metamodell mm_{Ref}

Engineering konzentriert, spielt die unterste Schicht hier eine untergeordnete Rolle. Ein weiteres Entwurfsprinzip für die Erstellung der konzeptuellen KOGITO-Metamodelle war die Unterteilung der Modellelemente auf allen drei Abstraktionsebenen in *Akteure*, *Prozesse* und *Entitäten*. Die für diese Arbeit relevanten Modelle werden in den nachstehenden Abschnitten detaillierter vorgestellt.

Das Business Reference Modell

Abbildung 5.2 stellt das *Business Reference Metamodell*, welches direkt aus der UMM [UN/01] (siehe Abschnitt 5.1.1) übernommen wurde, enthält eine Beschreibung der Geschäftsdomäne. In einer *Business Operations Map* wird zunächst die Branche in deren Umfeld eine neue Anwendung erstellt werden soll charakterisiert. Diese wird weiter in Geschäftsfelder (im Modell als *Business Area* bezeichnet) unterteilt. So ist beispielsweise eine Unterteilung der Branche „Zulieferindustrie“ in die Geschäftsfelder „Vertrieb“, „Personalwesen“, „Marketing“, etc. denkbar. Jedes Geschäftsfeld erfährt eine weitere Unterteilung in sog. Prozessfelder (*Process Areas*), die wiederum verwandte Geschäftsprozesse, wie sie in dem charakterisierten geschäftlichen Umfeld typischer Weise auftauchen, zusammenfasst.

Mit Hilfe des Business Reference Models wird bereits zu Beginn des Entwicklungsprozesses ein gemeinsames Vokabular für das fachliche Umfeld und eine erste Strukturierung der fachlichen Zusammenhänge festgelegt. Im Idealfall existieren bereits Referenzmodelle für bestimmte Branchen die zu Beginn eines Projektes direkt adaptiert werden können. Beispiele für solche existierenden Referenzmodelle sind die Porter Value Chain [Por80], das Supply Chain Operations Reference Model (SCOR) [SCO04] oder das VICS-Modell [VIC04] für den Einzelhandel.

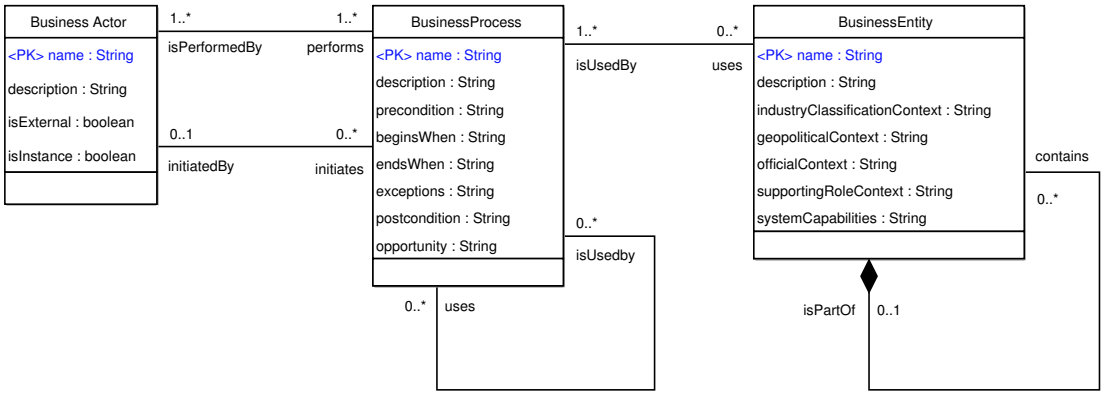


Abbildung 5.3: Das Business Requirements Metamodell mm_{BRM}

Das Business Requirements Modell

Ein Business Requirements Modell (BRM) enthält die Informationen, welche zu Beginn des Requirements Engineering Prozesses im Rahmen der Analyse des geschäftlichen Umfeldes einer Anwendung gesammelt werden. Im Gegensatz zum Business Reference Modell werden in diesem Modell bereits Abläufe und Zusammenhänge aus dem konkreten Umfeld der Anwendung aus fachlicher Sicht modelliert. In diesem Modell werden ausschließlich Konzepte des Anwendungsumfeldes verwendet. Abbildung 5.3 zeigt das vollständige Business Requirements Metamodell.

Die Menge der im Business Reference Model identifizierten Geschäftsprozesse kann im weiteren Verlauf der Analyse des Anwendungsumfeldes erweitert werden. Geschäftsprozesse werden textuell beschrieben. So enthält die Klasse Business Process des BRM Felder zur Spezifikation der Vor- und Nachbedingungen für die Ausführung eines Geschäftsprozesses, Kriterien wann der Prozess gestartet bzw. beendet wird, sowie für die textuelle Beschreibung des Prozesses, etc. Darüber hinaus können über uses-Assoziation Nutzungsbeziehungen zwischen Geschäftsprozessen spezifiziert werden.

In Instanzen der Klasse Business Actor des BRM-Metamodells finden sich Beschreibungen zu Aktoren in der Geschäftsumgebung. Aktoren initiieren Geschäftsprozesse oder sind an deren Ausführung beteiligt. Im BRM sind Aktoren immer Rollen oder handelnde Organe der fachlichen Welt, wie beispielsweise „Kunde“, „Lagerhaltung“ oder „Sachbearbeiter“. Ein Aktor kann Instanz eines anderen Aktors sein, falls er eine durch den anderen Aktor beschriebene Rolle einnimmt. Eine Unterscheidung zwischen Aktoren, die später durch technische Systeme realisiert werden und Personen bzw. Organisationen findet auf der Ebene des BRM noch nicht statt. Zu jedem Geschäftsprozess kann angegeben werden, durch welchen Aktor er angestoßen wird und welche Aktoren in seiner Ausführung involviert sind.

Informationen und Güter, die im Verlauf der Ausführung eines Geschäftsprozesses erzeugt, konsumiert oder verarbeitet werden, werden als Instanzen der Klasse Business Entity modelliert. So lassen sich jedem Geschäftsprozess eine Reihe von fachlichen Entitäten der Geschäftswelt zuordnen, die zu seiner Ausführung notwendig sind oder aber durch sie entstehen. Beispiele für solche Entitäten der Fachwelt sind die Entitäten „Rechnung“, „Bestellung“ aber auch physische Güter die produziert oder ausgetauscht werden. Mittels der isPartOf-Relation lassen sich Entitäten zudem hierarchisch gliedern.

Das BRM bietet also die Möglichkeit die fachlichen Anforderungen, wie sie aus der Sicht eines Auftraggebers für eine System-Entwicklung relevant erscheinen, strukturiert zu dokumentieren.

Das Requirements Analysis Modell

Im Requirements Analysis Modell (RAM) werden die Informationen des BRM weiter verfeinert und vervollständigt. Die generelle Trennung zwischen Aktoren, Prozessen und Entitäten bleibt erhalten, wobei Beziehungen zwischen diesen Elementen ebenfalls verfeinert werden. Abbildung 5.4 gibt einen Überblick über das Business Requirements Modell.

Aktoren des BRM werden weiter in Benutzer und Systemkomponenten unterteilt, wobei letztere eine hierarchische Struktur aufweisen können. Zusätzlich enthält die Klasse System Component des Metamodells ein Flag, welches angibt ob eine Systemkomponente Teil des zu entwickelnden Software-Systems ist. Hierdurch werden im RAM die Systemgrenzen und Schnittstellen des Systems mit der Umgebung festgelegt.

Geschäftsprozesse werden zu Instanzen der Klasse Business Collaboration verfeinert. Eine Business Collaboration besteht aus einer Menge von Business Activities und gerichteten Transitionen zwischen diesen Aktivitäten. Zwei Aktivitäten werden im Rahmen einer Business Collaboration sequentiell ausgeführt, falls sie durch eine entsprechend gerichtete Transition verbunden sind. Business Activities untergliedern sich wiederum in

Control Activities Mögliche Arten von Kontrollaktivitäten sind Verzweigungen, das Aufspalten in parallel ablaufende Aktivitätssequenzen oder das Zusammenführen paralleler Sequenzen. Darüber hinaus existieren die Kontrollaktivitäten „Startaktivität“, die den Ausgangspunkt einer Business Collaboration markiert, und „Endaktivität“, die das Ende einer Kollaboration markiert.

Transaction Activities Diese Art von Aktivitäten beschreibt die Interaktion von zwei Systemkomponenten. Jeder Transaction Activity wird je eine, die Interaktion initiierende (requestor), und eine reagierende Komponente (responder) zugeordnet.

Internal Business Activity Finden Verarbeitungsschritte innerhalb einer Systemkomponente statt, so werden diese als Internal Business Activities modelliert. Jeder solchen Aktivität wird eine Systemkomponente zugeordnet, durch die die jeweilige Berechnung erfolgt.

User Interaction Activity Dieser Typ von Aktivität steht für Interaktionen eines Benutzers mit einer Systemkomponente. Dementsprechend wird jeder User Interaction Activity ein Aktor des Typs User und eine involvierte Systemkomponente zugeordnet.

Zusätzlich wird jeder Business Activity eine Reihe von produzierten (produces), benötigten (requires) und benutzten (uses) Business Entity Instances zugeordnet. Eine Business Entity Instance repräsentiert eine Instanz einer Entität aus dem BRM.

Entitäten selbst werden zu konzeptuellen Datenmodellen (Business Data Model) verfeinert. Im RAM-Metamodell wird ein konzeptuelles Datenmodell durch ein objektorientiertes Klassenmodell spezifiziert. Die Klassen und Assoziationen zur Definition des Business Data Models entsprechen genau den, in Abschnitt 3.1.2 (s. S. 52) eingeführten, objektorientierten Metamodellen. Mit Hilfe konzeptueller Datenmodelle lassen sich Informationen, die durch das Software-System verarbeitet werden, strukturieren ohne bereits konkrete Datenaustausch- oder Speicherformate festlegen zu müssen. Die hierarchische Struktur der Business Enities im BRM schägt sich in einer hierarchischen Strukturierung des konzeptuellen Datenmodells durch Kompositionsbeziehungen nieder.

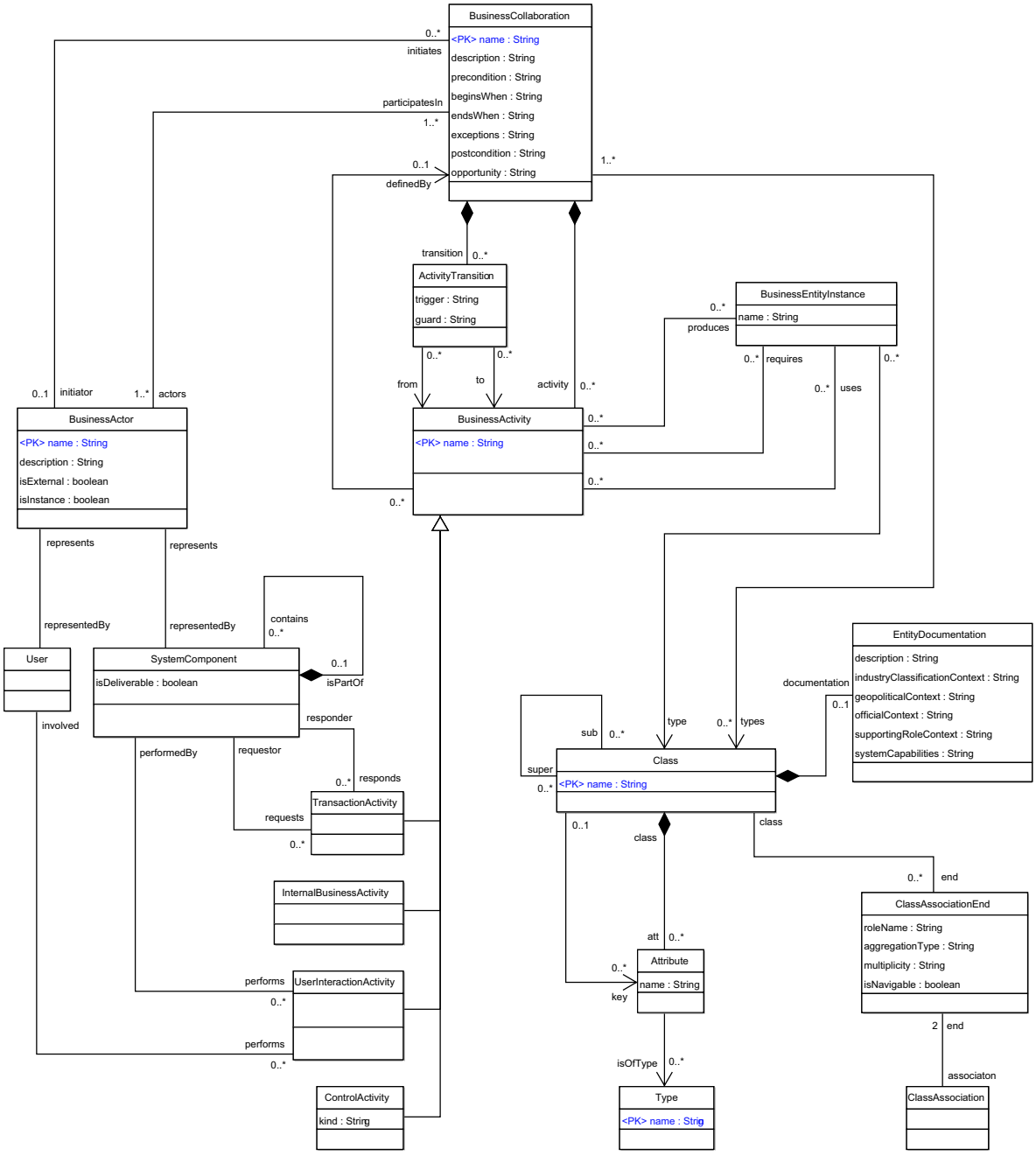


Abbildung 5.4: Das Requirements Analysis Metamodell mm_{RAM}

5.1.3 KOGITO-Entwicklungsartefakte

Für den Anwender der KOGITO-Methodik bleiben die konzeptuellen Modelle selbst unsichtbar. Eine direkte Bearbeitung dieser Modelle wäre unergonomisch und nur wenig intuitiv. Statt dessen bietet KOGITO eine Reihe von Artefakttypen, mit denen sich *Sichten* auf ein konzeptuelle Modell modellieren lassen (siehe Definition 2.1.5, S. 25).

Für jedes konzeptuelle Modell und das Business Reference Modell existiert eine Reihe von Artefakttypen, mit denen sich einzelne Aspekte des konzeptuellen Modells auf einer bestimmten Abstraktionsstufe beschreiben lassen. Die verwendeten Beschreibungstechniken lassen sich in zwei Kategorien unterteilen: zum einen existieren Formulare, in denen strukturiert Text hinterlegt werden kann und zum anderen existieren eine Reihe graphischer Beschreibungstechniken, die zumeist aus der UML stammen. Diese Art der Dokumentation von Anforderungen stößt gemäß einer in [ABG+03] vorgestellten Studie auf die höchste Benutzerakzeptanz. So wurde im Rahmen der Studie der Unterstützungsgrad von natürlicher Sprache für das Requirements Engineering mit 76%, der der UML mit 100%, bewertet.

Formulare stoßen vor allem in den frühen Phasen des Requirements Engineering bei den Benutzern, die sich zu diesem Zeitpunkt in erster Linie aus Experten des fachlichen Umfeldes der Anwendung zusammensetzen, auf höhere Akzeptanz als in der Software-Technik verwendete graphische Notationen. Dennoch wird durch die in den Formularen vordefinierte Struktur und durch die Auswahl der Formulararten bereits in diesen frühen Phasen sichergestellt, dass Informationen strukturiert gesammelt und möglichst vollständig erhoben werden.

Graphische Beschreibungstechniken erlauben im Gegensatz zu strukturiertem, natürlichsprachlichem Text eine wesentlich präzisere Modellierung von Sachverhalten. Im Rahmen der KOGITO-Methodik werden graphische Beschreibungstechniken in erster Linie für den Aufbau des Requirements Analysis Models und seiner Verfeinerungen verwendet. Aufgrund der hohen Akzeptanz der UML-Beschreibungstechniken werden diese auch in KOGITO verwendet. Allerdings werden UML-Diagramme (im Gegensatz zu EDOC) unabhängig von der UML-Semantik verwendet, da dies mit einer Erweiterung des ohnehin sehr unübersichtlichen und semantisch nicht immer eindeutig fundierten UML Metamodells durch verbunden wäre.

Entwicklungsartefakte des Business Reference Modells

Das Business Reference Modell wird anhand einer Reihe von Formularen erstellt, die im Wesentlichen aus der UMM-Methodik übernommen wurden.Im Einzelnen sind dies:

Business Stakeholder Worksheet In Formularen dieses Typs werden Informationen zu sog. *Stakeholdern* eingetragen. Dies sind Personen oder Gruppen von Personen, die einen Bezug zu der entwickelnden Anwendung oder deren direktem Umfeld hat.

Business Reference Model Worksheet Dieser Formulartyp dient zur Beschreibung eines Referenzmodells für die *Anwendungsdomäne*. Es enthält neben Feldern für die textuelle Beschreibung des Industriesegments und zur Abgrenzung gegenüber anderen Referenzmodellen Verweise auf eine Reihe relevanter Business Area Worksheets (siehe unten).

Business Area Worksheet Durch Formulare dieses Typs werden *Geschäftsfelder* innerhalb einer Anwendungsdomäne textuell beschrieben. Es sieht weiterhin Verweise auf Business Stakeholder Formulare vor, in denen Personen oder Gruppen charakterisiert sind, die innerhalb des Geschäftsfeldes als Handelnde oder Verantwortliche eine Rolle spielen.

Process Area Worksheet Diese Art von Formularen erlaubt die textuelle Beschreibung eines *Prozessfeldes* als Teil eines Geschäftsfeldes. Ein Process Area Worksheet kann eine Reihe von Geschäftsprozessen anführen, aus denen sich das Prozessfeld zusammensetzt.

Business Process Description Worksheet Mit Formularen dieses Typs wird jeweils ein *Geschäftsprozess* beschrieben. Das Formular enthält neben der Beschreibung des eigentlichen Prozesses auch Textfelder zur Angabe seiner Ziele, Vor- und Nachbedingungen, initierende Aktoren, usw. Tabelle 5.1 zeigt ein Beispiel für ein Artefakt dieses Typs.

Business Process Description Worksheet	
Name:	Create Order
Precondition:	A *sales manager* is available
Begins When:	A *customer* contacts a sales manager
Definition:	After the *sales manager* has been contacted by the *customer* he collects information about the *customer* and the desired type and amount of *products*. All information must be added to a newly created *order*.
Ends When:	The *order* is complete
Exceptions:	• No *customer information* is available • The desired *product* is unknown • The *customer* withdraws his *order*
Postcondition:	A new *order* is created.
Actors:	• Customer • Sales Manager
Involved Entities:	• Order
Subprocesses:	(none)

Tabelle 5.1: Beispiel für ein Artefakt des Typs „Business Process Description Worksheet“

Beziehungen zwischen den einzelnen Artefakttypen ergeben sich durch Verweise auf Elemente mit gleichen Namen. Beispielsweise wird durch die Angabe eines Geschäftsprozesses in der Beschreibung eines Prozessfeldes implizit eine Beziehung zu dem entsprechenden Business Process Description Worksheet hergestellt, welches einen Geschäftsprozess mit dem gleichen Namen beschreibt. Abbildung 5.5 skizziert das Produktmodell der Entwicklungsartefakte für die Beschreibung des Business Reference Modells.

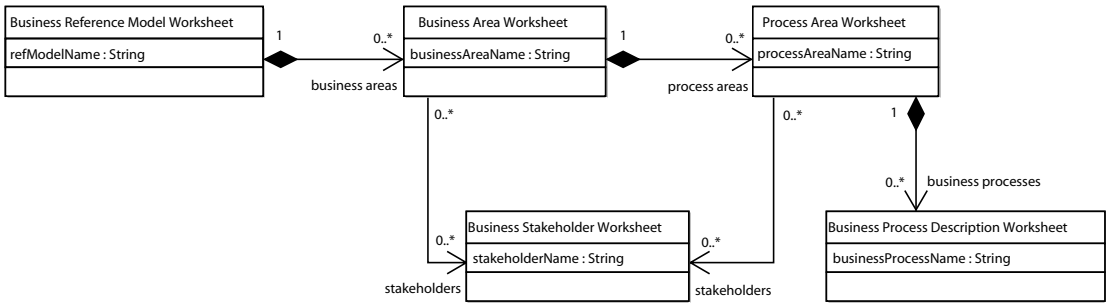


Abbildung 5.5: Das Produktmodell der Entwicklungsartefakte für die Beschreibung des Business Reference Modells

Entwicklungsartefakte für das Business Requirements Modell

Auch für die Modellierung des BRM kommen in erster Linie Formulare zum Einsatz. Im einzelnen existieren vier Artefakttypen zur Beschreibung des BRM. Diese werden im Folgenden lediglich aufgelistet, eine detaillierte Beschreibung findet sich in [MS03c]

Business Process Description Worksheet Dieser Artefakttyp wird auch für die Modellierung des Business Reference Modells verwendet. Für die Modellierung des Business Requirements Modells können ggf. zusätzliche solcher Formulare erstellt werden, in denen für die Anwendung spezifische Geschäftsprozesse charakterisiert werden.

Business Entity Worksheet Informationen und Güter der Geschäftswelt werden mit Formularen dieses Typs in Form von strukturiertem Text beschrieben. Ein solches Formular kann unter anderem Verweise auf Geschäftsprozesse haben, in deren Kontext die jeweilige Entität verwendet wird.

Business Actor Worksheet Das Formular dieses Typs dienen der Beschreibung von Aktoren der Geschäftswelt aus fachlicher Sicht. Weiterhin kann festgelegt werden, ob der jeweilige Aktor Teil der eigenen Organisation ist und ob er eine Instanz eines anderen Aktors darstellt.

Business Process Use Case Diagram UML Use Case Diagramme werden verwendet um Geschäftsprozesse, Geschäftsentitäten und Aktoren sowie deren Beziehungen untereinander graphisch zu modellieren.

Entwicklungsartefakte für das Requirements Analysis Modell

Für den Aufbau des Requirements Analysis Modells sind in KOGITO eine Reihe von Formularen und graphischen Beschreibungstechniken vorgesehen. Im Folgenden wird lediglich eine Auswahl der verwendeten Artefakttypen vorgestellt, eine vollständige Übersicht kann [MS03c] entnommen werden.

Business Data Model Diagram Ein Business Data Model Diagram modelliert die Struktur von Informationen die von Business Actors erzeugt oder verarbeitet werden. Als Notation werden UML-Klassendiagramme verwendet. Die Notation ist in [OMG02c] spezifiziert.

Business E/R-Diagram Als Alternative zum Business Data Model können Entity-Relationship-Diagramme als Notation zur Modellierung der Geschäftsentitäten verwendet werden.

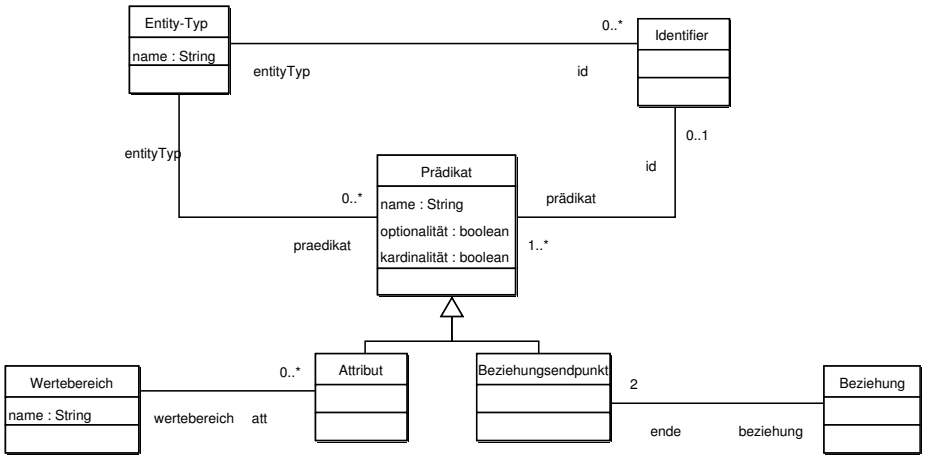


Abbildung 5.6: Das Metamodell mm_{ER} zur Spezifikation der abstrakten Syntax von Entity-Relationship-Diagrammen

Abbildung 5.6 zeigt ein einfaches UML-Klassendiagramm, durch welches die abstrakte Syntax von Entity-Relationship-Diagrammen in Form eines Metamodells spezifiziert ist. Das Modell ist im wesentlichen [Rit02] entnommen.

In Tabelle 5.2 wird die konkrete Syntax textuell spezifiziert. Ähnliche Notationen sowie eine textuell spezifizierte Semantik für Entity-Relationship-Diagramme finden sich beispielsweise in [KE97, Eng93].

Business Activity Diagram Dieser Artefakttyp verwendet ein Profil für UML Aktivitätsdiagramme als Notation. Aktivitäten der Aktivitätsdiagramme werden auf Business Activities des RAM abgebildet. Mit Hilfe der Stereotypen (Business Activity, User Interaction Activity, Business Transaction Activity und Internal Business Activity wird festgelegt, welche Art von Aktivität ein UML-Aktivitätssymbol repräsentiert. Aktivitäten können anhand von Objektflusspfeilen mit Objekten die sie erzeugen oder verarbeiten verbunden werden. Diese Objekte sind jeweils Instanzen der im Business Data Model definierten Geschäftsentitäten.

5.2 Transformation objektorientierter konzeptueller Modelle mit BOTL

Zur Veranschaulichung wird nun exemplarisch eine vergleichsweise einfache Abbildung des in KOGITO verwendeten Business Requirements Modells (BRM) auf das Requirements Analysis Modell (RAM) untersucht. Wie in Abbildung 5.7 skizziert, wird zunächst eine Abstraktionsabbildung zwischen den Modelltypen in Form des Regelwerks R_a angegeben. Hierdurch wird festgelegt, wann ein Business Requirements Modell eine Verfeinerung eines bestehenden Requirements Analysis Modells darstellt. Auf Basis dieser Abstraktionsabbildung wird im Anschluss das Regelwerk R_{BRM} als mögliche Verfeinerungsabbildung vorgestellt. Da es sich bei R_{BRM} genau um das Umkehrregelwerk von R_a handelt, muss lediglich die Bijektiviät dieses Regelwerks nachgewiesen werden, um zu zeigen, dass R_{BRM} eine gültige Verfeinerungsabbildung darstellt.

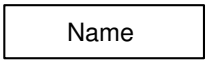
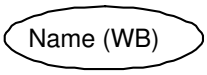
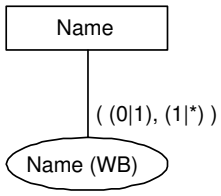
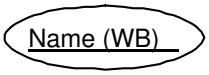
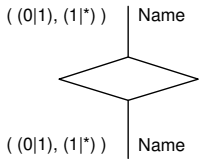
Symbol:	Bedeutung / Abbildung auf die abstrakte Syntax:
	Ein *Entitäts-Typ* wird durch ein Rechteck dargestellt, in dessen Mitte der Name des Entitätstyps steht. Innerhalb der abstrakten Syntax wird ein Entitätstyp als als Objekt der Klasse Entity-Typ dargestellt.
	Ein *Attribut* wird durch ein Oval dargestellt, in dessen Mitte der Name des Attributs steht. Hinter dem Namen steht in Klammern der Wertebereich für die möglichen Werte des Attributes.
	Die *Zuordnung von Attributen zu Entitäten* erfolgt durch eine Linie an der die Multitplizität des Attributs vermerkt ist. Innerhalb von E/R-Diagrammen sind lediglich 0 oder 1 als mögliche Untergrenzen und 1 oder $*$ (entspricht ∞) als Obergrenzen zugelassen. Gehört ein Attribut zu einer Entität, so wird dies durch einen Link zwischen den beiden entsprechenden Objekten der abstrakten Syntax modelliert. Ist die Multiplizitätsuntergrenze des Attributs 0 so hat das Attribut optional des Objekts der abstrakten Syntax den Wert true, andernfalls den Wert false. Ist die Multiplizitätsobergrenze des Attributs $*$ so hat das Attribut mehrfach des Objekts der abstrakten Syntax den Wert true, andernfalls den Wert false.
	Ein *Identifikator* ist ein Attribut dessen Namen und Wertebereich unterstrichen sind. Identifikatoren entsprechen Primärschlüsseln, anhand derer sich Entitäten eindeutig identifizieren lassen. In der abstrakten Syntax entspricht ein Identifikator einer Instanz der Klasse Identifier, die jeweils über einen Link mit dem durch das Entitätssymbol repräsentierte Entity-Typ-Objekt und dem entsprechenden Attribut verbunden ist.
	Eine *Beziehung* wird durch eine Raute dargestellt, die über Linien mit genau zwei Entitätstypen verbunden ist. An jeder Linie findet sich ein *Rollenname* und eine *Multiplitzität* (siehe oben). Ein Beziehung entspricht in der abstrakten Syntax einem Objekt der Klasse Beziehung das mit genau zwei Objekten der Klasse Beziehungsendpunkt verbunden ist. Die Rollennamen und die Multiplizitäten finden sich analog wie bei den Instanzen der Klasse Attribut in den Attributen der Beziehungsendpunkt-Objekte.

Tabelle 5.2: Konkrete Syntax der in KOGITO verwendeten Entity-Relationship-Digagramme

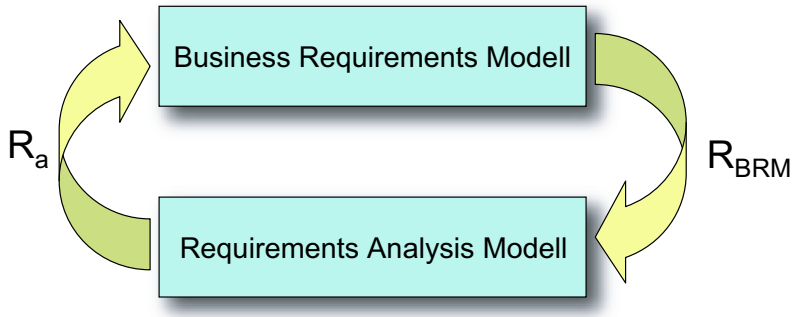


Abbildung 5.7: Abstraktions- und Verfeinerungsabbildung zwischen konzeptuellen KOGITO-Modellen

5.2.1 Grundsätzliches

Die Verwendung konzeptueller Modelle auf verschiedenen Abstraktionsebenen, wie sie in Abschnitt 2.2.3 diskutiert wurde, deckt sich in weiten Teilen mit den Konzepten der Model Driven Architecture (MDA) (siehe Abschnitt 1.3.3, S. 9). Sie ermöglicht zum einen die Wiederverwendung abstrakter Modelle, zum anderen können dem Entwickler durch automatisierte Abbildungen Design-Entscheidungen abgenommen werden, indem durch die Transformationsspezifikation die Verwendung bewährter Entwurfsmuster [GHJV94] sichergestellt wird.

Das einzige bislang existierende Beispiel für eine kommerzielle Werkzeugunterstützung eines MDA-Entwicklungsansatzes ist das Werkzeug OptimalJ des Herstellers Comuware [Com04]. Dieses Werkzeug setzt bei der Verfeinerung eines plattformunabhängigen UML-Modells zu einem EJB-spezifischen UML-Modell die Entwurfsmuster des „Sun J2EE Pattern Catalog“ um [Sun04, Kui03]. Wie bei allen MDA-basierten Ansätzen wird für die Spezifikation von Modelltransformationen eine geeignete Sprache benötigt. Die Entwicklung einer solchen Spezifikationssprache ist auch das Ziel der QVT-Initiative der OMG (siehe Abschnitt 1.3.4). Die Konzepte der intern von OptimalJ verwendeten Transformationssprache finden sich auch in einer gemeinsamen QVT-Einreichung der Firmen Compuserve und SUN wieder [CM03].

Allen QVT-Einreichungen ist gemein, dass sie keine Möglichkeiten zur formalen Verfifiktion der Metamodellkonformität von Transformationsspezifikationen vorsehen. Da es sich zudem bei den meisten Ansätzen um textbasierte Sprachen oder Mischformen handelt, sind die bisher verfügbaren Sprachen kaum geeignet um nachprüfbar konsistente Verfeinerungsabbildungen objektorientierter, konzeptueller Modelle zu spezifizieren. Aus diesem Grund wird im Folgenden BOTL für die Spezifikation von Verfeinerungsabbildungen konzeptueller Modelle eingeführt. In [MB03] findet sich eine detaillierte Beschreibung wie sich BOTL innerhalb eines MDA-basierten Entwicklungsansatzes einsetzen lässt. Die wesentlichen Aspekte, die durch Spezifikationen von Modelltransformationen für konzeptuelle Modelle abgedeckt werden müssen sind im Einzelnen:

1. Durch eine *Abstraktionsabbildung* zwischen Instanzen zweier konzeptueller Modelle wird festgelegt, wann eine Instanz des einen konzeptuellen Modells eine gültige Verfeinerung des anderen darstellt.

2. Ist eine Abstraktionsabbildung gegeben, so können verschiedene *Verfeinerungsabbildungen* definiert werden. Für den Spezialfall, dass eine solche Verfeinerungsabbildung bijektiv ist, ist offensichtlich sichergestellt, dass die Verfeinerungsabbildung einer gültige Verfeinerung bezüglich der gewählten Abstraktion darstellt (siehe auch Abschnitt 2.2.3, S. 38 ff).

BOTL kann direkt für die Transformation objektorientierter, konzeptueller Modelle verwendet werden: Sei $mm_a \in \mathbb{MM}$ ein konzeptuelles Metamodell $mm_r \in \mathbb{MM}$ ein konzeptuelles Metamodell auf einer niedrigeren Abstraktionsebene. Damit durch ein BOTL-Regelwerk eine metamodellkonforme Transformation (siehe Definition 2.2.2 S. 40) von Quellmodellen aus $\mathbb{MM}_{mm_a}$ in Zielmodelle aus $\mathbb{MM}_r$ spezifiziert wird, müssen alle Regeln aus R_r das Metamodell mm_a als Quellmetamodell und mm_r als Zielmetamodell haben. Weiterhin muss das Regelwerk metamodellkonform sein, damit sichergestellt ist, dass es ausschließlich Modelle erzeugt die konform zum Metamodell mm_r sind. Formal bedeutet dies im Einzelnen:

(i) $MM_R^0 = \{mm_a\}$

(ii) $mm_R^1 = mm_r$

(iii) $mmConform(R)$

Gemäß Definition 2.2.5 ist eine Abbildung $refine$ zwischen konzeptuellen Modellen eine Verfeinerungsabbildung bezüglich einer Abstraktionsabbildung $abstract$, falls gilt:

$$\forall m_a \in \overline{\mathbb{M}}_{mm_a} : abstract(refine(m_a)) = m_a$$

Werden die Modelltransformationen $abstract$ und $refine$ in Form von BOTL-Regelwerken R_a und R_r spezifiziert, so entspricht deren Anwendung jeweils der Anwendung der jeweiligen BOTL-Regelwerke R_a bzw. R_r. Dementsprechend muss für eine korrekte Verfeinerungstransformation R_r für objektorientierte Modelle bei der Anwendung von BOTL-Regelwerken gelten:

$$\forall m_a \in \mathbb{M}_{mm_a} : transform(transform(\{m_a\}, R_r), R_a) = m_a \tag{5.1}$$

Gemäß Definition 4.3.2 (s. S. 168) ist diese Eigenschaft sicherlich für den Spezialfall erfüllt, in dem $R_a = R_r^{-1}$ gilt und R_a ein streng bijektives Regelwerk ist.

5.2.2 Eine Abstraktionsabbildung für Requirements Analysis Modelle

Abbildung 5.8 stellt graphisch ein Business Requirements Modell m_{BRM} dar, welches lediglich einen einzigen Geschäftsprozess „Create Order“ umfasst. Der Prozess wird durch einen Aktor „Customer“ angestoßen, weiterhin werden der Aktor „SalesManger“ und die Geschäftsentität „Order“ von diese Geschäftsprozess referenziert. Das Metamodell mm_{BRM} des BRM ist in Abbildung 5.3 dargestellt.

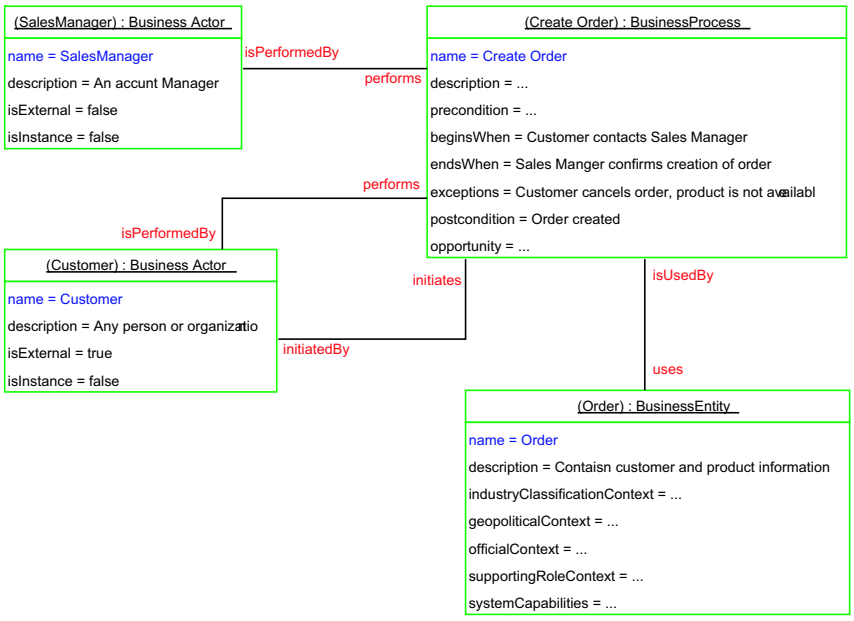


Abbildung 5.8: Beispiel für ein gültiges Business Requirements Modell m_{BRM}

Um ein BRM mittels einer Verfeinerungsabbildung automatisch in ein RAM überführen zu können muss zunächst festgelegt sein, welche Instanzen des RAM-Metamodells gültige Verfeinerungen eines BRM-Modells sind. Hierzu wird im Folgenden die Abstraktionsabbildung gemäß Definition 2.2.3 zwischen RAMs und BRMs definiert.

Die gewünschte Abstraktionsabbildung wird durch das Regelwerk $R_a = \{r_{a1}, r_{a2}, r_{a3}, r_{a4}, r_{a5}\}$ festgelegt. Die einzelnen Regeln des Regelwerks sind in den Abbildungen 5.9 bis 5.13 dargestellt.

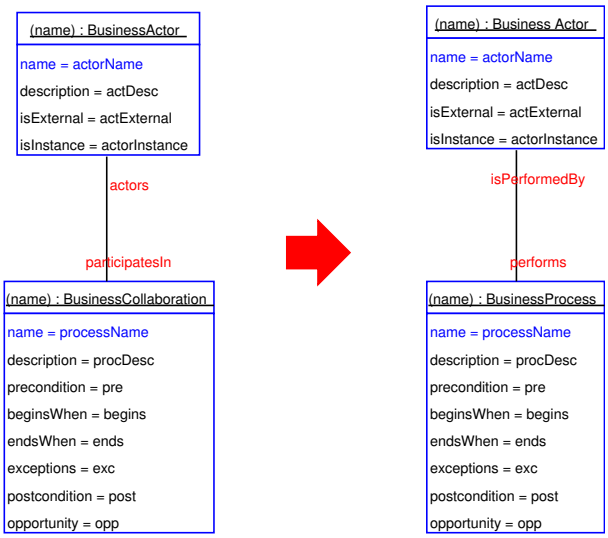


Abbildung 5.9: Abbildungsregel r_{a1}

Regel r_{a1} erzeugt für jedes Paar aus Business Actor und Business Collaboration im RAM einen Business Process und einen Business Actor im BRM. Sämtliche Daten zur Beschreibung der Prozesse und Aktoren werden hierbei ebenfalls kopiert.

Regel r_{a2} erzeugt zu allen Klassen im Business Data Model des BRM die mit einer Beschreibung (Business Entity Documentation) versehen sind eine Business Entity im BRM. Die Daten der Business Entity Description werden in die Attribute der Business Entity kopiert. Für jede Business Collaboration, in der die entsprechende Klasse verwendet wird, wird eine „uses“-Assoziation zu dem Geschäftsprozess im BRM der durch die Business Collaboration verfeinert wird erzeugt.

Zusätzlich zur „uses“-Assoziation werden in Regel r_{a3} die „initiates“-Beziehungen zwischen Geschäftsprozessen im BRM aus den entsprechenden Beziehungen im RAM erzeugt.

Regel r_{a4} erzeugt zu jeder Business Activity einer Business Collaboration im RAM die durch eine weitere Business Collaboration verfeinert wird eine “uses“-Assoziation zwischen den beiden zugehörigen Geschäftsprozessen im BRM.

Die in Abbildung 5.13 dargestellte Regel r_{a5} sucht im RAM nach Klassen des Business Data Modells die in einer Kompositionsbeziehung zueinander stehen und beide über Entity Documentation verfügen. Diese Beziehung wird auf eine „isPartOf“-Beziehung zwischen den zugehörigen Business Enities des BRM abgebildet.

Damit R_a eine gültige Abstraktionsabbildung darstellt muss zunächst sichergestellt werden, dass das Regelwerk R_a metamodellkonform ist. Soll R_a^{-1} als Verfeinerungsabbildung verwendet werden, so muss auch deren Metamodellkonformität bewiesen werden. Die Metamodellkonformität von R_a und R_a^{-1} lässt sich mit den in Abschnitt 4 vorgestellten Techniken nachweisen. Dieser Nachweis wird an dieser Stelle nicht nachvollzogen um den Rahmen der Arbeit nicht zu sprengen. Eine detailliertes Beispiel für den Nachweise der Metamodellkonformität eines Regelwerks findet sich in Anhang B.

Anmerkung: Innerhalb des Regelwerks R_a schreiben mehrere Objektvariablen gleichen Typs aus verschiedenen Regeln denselben Wert in Attribute des Zielmodells, was die Konfliktfreiheit der Objektvariablen des Regelwerks nicht offensichtlich erscheinen lässt. Der für den Nachweis der Konfliktfreiheit der einzelnen Objektvariablen kann jedoch unter Zuhilfenahme von Satz 4.1.5 erbracht werden. Ein Nachweis mit Hilfe des wesentlich einfacher zu verifzierenden Lemmas 4.1.6 ist in die-

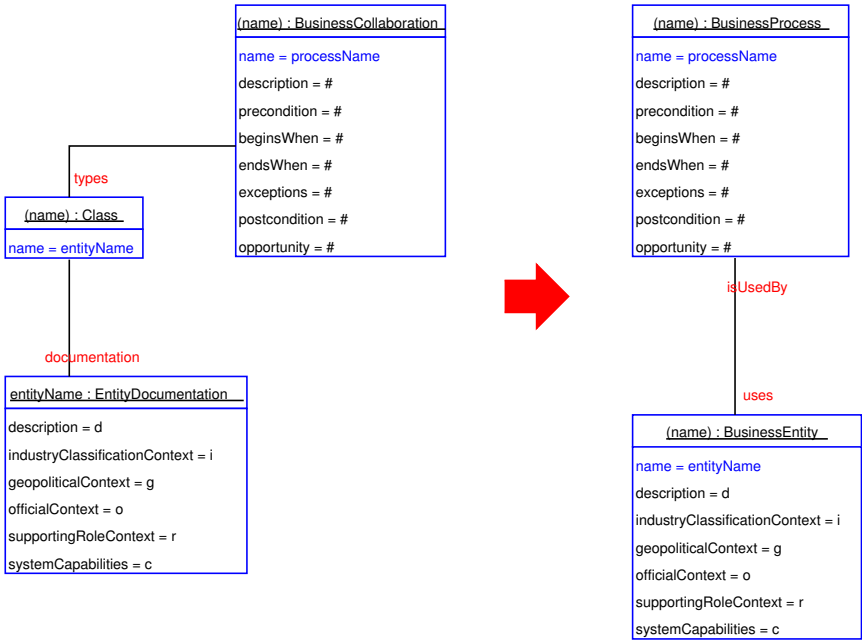


Abbildung 5.10: Abbildungsregel r_{a2}

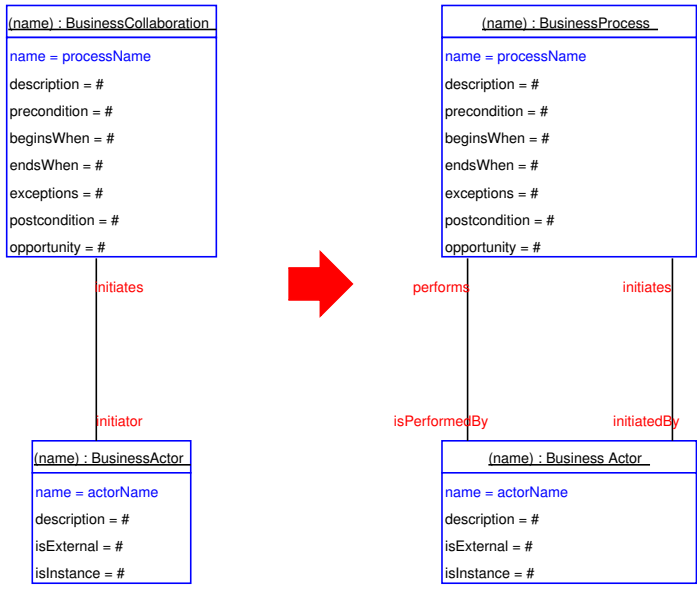


Abbildung 5.11: Abbildungsregel r_{a3}

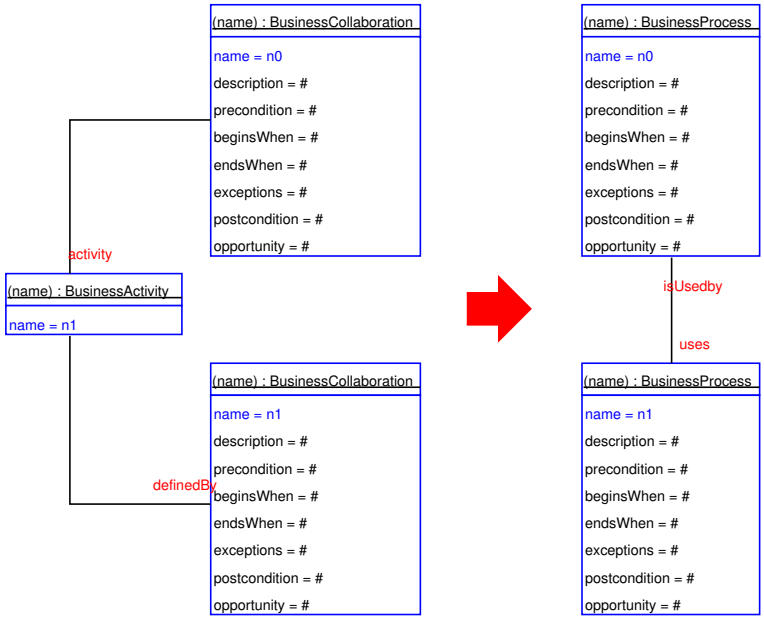


Abbildung 5.12: Abbildungsregel r_{a4}

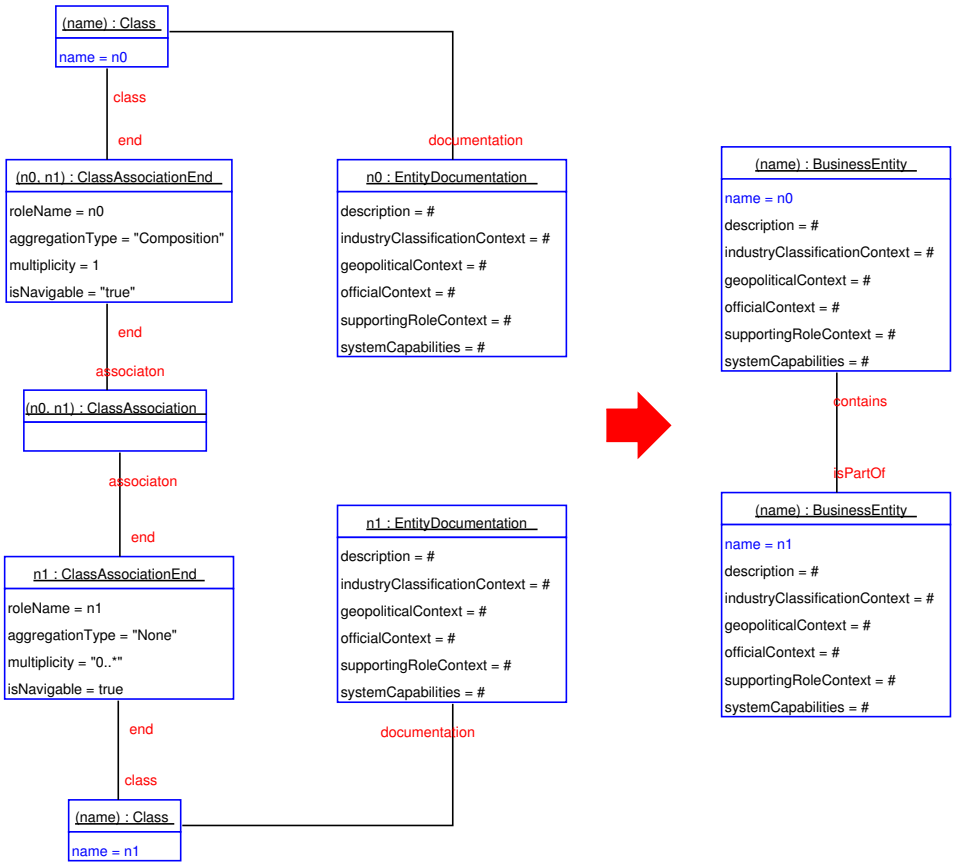


Abbildung 5.13: Abbildungsregel r_{a5}

sem Fall für die meisten Objektvariablen nicht möglich.

Mit Hilfe der festgelegten Abstraktionsabbildung ist es möglich zu einem späteren Zeitpunkt, nachdem das RAM um Informationen aus zusätzlichen Artefakten angereichert wurde, festzustellen ob das so erweiterte Modell immer noch eine gültige Verfeinerung des ursprünglichen BRMs bezüglich der gewählten Abstraktion darstellt.

5.2.3 Eine Verfeinerungsabbildung für Business Requirements Modelle

Mit Hilfe der so definierten Abstraktionsabbildung R_a lässt sich nun ermitteln, ob eine Abbildung von BRM-Modellen auf RAM-Modelle eine gültige Verfeinerung bezüglich dieser Abstraktion darstellt.

Definition 2.2.5 legt fest, wann eine durch ein Regelwerk R_{BRM} definierte Abbildung eine gültige Verfeinerung darstellt. Für das Regelwerk R_{BRM} muss demnach gemäß (5.1) (S. 194) gelten:

$$\forall m \in \mathbb{M}_{mm_{BRM}} : refines_{R_a}(transform(m, R_{BRM}), m)$$
$$\overset{\text{Def. 2.2.5}}{\Leftrightarrow} \forall m \in \mathbb{M}_{mm_{BRM}} : transform(transform(m, R_{BRM}), R_a) = m$$

Diese Eigenschaft gilt insbesonders dann, falls das Regelwerk R_{BRM} streng bijektiv bezüglich seines einzigen Quellmetamodells ist und $R_{BRM}^{-1} = R_a$ gilt.

In diesem Fall kann die Umkehrabbildung der Abstraktionsabbildung R_a als mögliche Verfeinerungsabbildung verwendet werden. Dies ist sicherlich nicht für jede Verfeinerungsabbildung der Fall. Würde die Abstraktionsabbildung Elemente des Typs „User“ und des Typs „System Component“ im RAM auf Aktoren des BRM abbilden, wäre die gewählte Abstraktionsabbildung mit Sicherheit nicht mehr bijektiv, da die Umkehrabbildung für jeden Aktor ein Element vom Typ „User“ und zusätzlich eines vom Typ „System Component“ erzeugen würde. ließe sich sicherlich keine bijektive Verfeinerungsabbildung finden. Generell sind zu einer Abstraktionsabbildung eine Reihe verschiedener Verfeinerungsabbildungen denkbar. Beispielsweise könnte eine mögliche Verfeinerungsabbildung Geschäftsprozesse mit ihr bekanntem Namen in bereits vordefinierte Business Collaboration transformieren. Im Folgenden sei $R_{BRM} = \{r_b1, \ldots, r_b5\} = \{r_a1^{-1}, \ldots r_{a5}^{-1}\}$ das Umkehrregelwerk von R_a. Im vorliegenden Fall wird das Regelwerk R_{BRM} als möglich Verfeinerungsabbildung zwischen BRM und RAM verwendet.

Um die Bijektivität von R_{BRM} bezüglich des Quellmetamodells mm_{BRM} nachzuweisen genügt es zu zeigen, dass das Regelwerk streng match-bijektiv gemäß Definition 4.3.5 ist. Die Metamodellkonformität des Regelwerks und seines Umkehrregelwerks vorausgesetzt, bleibt hierzu noch zu zeigen:

- Es existiert eine *map*-Abbildung mit den in Definition 4.3.5 (iii) geforderten Eigenschaften.
- Es gilt: $\bigcup_{\mathfrak{m} \atop mfm \in srcMatches(M,R')} mfm|_{mf} = m \in M : m_{mm} = mm_{BRM}$ (Def. 4.3.5 (iv)).

Die *map*-Abbildung wird nun folgendermaßen angegeben:

$$\forall m \in \mathbb{M}_{mm_{BRM}}, r \in R_a, mfm \in MFM(m, r|_{mv_0}) :$$
$$map(mfm_i) = \text{Der Match im Zielmodellfragment von } mft(mfm_i, r)$$

Definition 4.3.5 (4.30) gilt offensichtlich:

$$mft(mfm_i, r) = map(mfm_i)|_{mf}$$

Für eine Regel $r \in R_a$ sei GL_r das Gleichungssystem gemäß Definition 3.5.4. Einsetzen der Lösung von GL_r in das Gleichungssystem $GL_{r^{-1}}$ liefert für sämtliche Regeln Aussagen der Form

$$0 = 0$$

Beispielsweise ergeben sich für das Gleichungssystem GL_{ra1} Gleichungen der Art:

$$\begin{aligned} mfm^0_\mu|_{match_o}(bav0).name &= actorName \\ mfm^1_\mu|_{match_o}(bav1)|_{oi} &= pK(\{(name, mfm^1_\mu|_{match_o}(bav1).name)\}) \\ mfm^1_\mu|_{match_o}(bav1).name &= actorName \\ &\dots \end{aligned}$$

mit der Lösung:

$$\begin{aligned} mfm^1_\mu|_{match_o}(bav1)|_{oi} &= pK(\{(name, mfm^0_\mu|_{match_o}(bav0).name)\}) \\ mfm^1_\mu|_{match_o}(bav1).name &= mfm^0_\mu|_{match_o}(bav0).name \\ &\dots \end{aligned}$$

Umgekehrt erhält man für $GL_{ra1^{-1}}$:

$$\begin{aligned} mfm^0_\mu|_{match_o}(bav1).name &= actorName \\ mfm^1_\mu|_{match_o}(bav0)|_{oi} &= pK(\{(name, mfm^1_\mu|_{match_o}(bav0).name)\}) \\ mfm^1_\mu|_{match_o}(bav0).name &= actorName \\ &\dots \end{aligned}$$

mit der Lösung:

$$\begin{aligned} mfm^1_\mu|_{match_o}(bav0)|_{oi} &= pK(\{(name, mfm^0_\mu|_{match_o}(bav1).name)\}) \\ mfm^1_\mu|_{match_o}(bav0).name &= mfm^0_\mu|_{match_o}(bav1).name \\ &\dots \end{aligned}$$

Alle Umkehrregeln von R_{BRM} erzeugen folglich Objekte mit denselben Identifikator- und Attributwerten wie die ursprünglichen Regeln, wenn sie auf ein Fragment matchen das die ursprüngliche Regel erzeugt hat. D.h. es gilt Aussage (4.31) aus Definition 4.3.5:

$$mft(map(mfm_i), r^{-1}) = mfm_i|_{mf}$$

Es bleibt nun zu zeigen, dass *map* eine bijektive Abbildung ist, d.h. jede Umkehrregel darf nur genau auf die Fragmente matchen die durch eine Anwendung der Ursprungsregel erzeugt wurden.

Regel r_{b1} erzeugt ein Objekt der Klasse Business Collaboration und ein Objekt der Klasse Business Actor. Lediglich Regel r_{a3} erzeugt zwei Objekte desselben Typs. Da r_{a3} jedoch die einzige Regel ist, in der die „initiates“-Assoziation vorkommt, kann diese Regel auf keine von r_{a1} erzeugten Fragmente matchen, die sie nicht auch selbst erzeugt hat.

Regel r_{b2} erzeugt je ein Objekt der Klasse Business Collaboration, Class und Entity Documentation. Die „types“-Assoziation, welche die beiden Objektvariablen vom Typ Business Collaboration und Class verbindet, kommt in keiner anderen Regel vor. Folglich kann nur die Umkehrregel von r_{a2} auf die von ihr erzeugten Fragmente matchen.

Regel r_{b3} erzeugt ein Objekt der Klasse Business Collaboration und ein Objekt der Klasse Business Actor. Die „initiates“-Assoziation, welche die beiden Objektvariablen verbindet, kommt in keiner anderen Regel vor, folglich kann nur die Umkehrregel von r_{a3} auf die von ihr erzeugten Fragmente matchen.

Regel r_{b4} erzeugt zwei Objekte der Klasse Business Business Collaboration und ein Objekt der Klasse Business Activity. In keiner anderen Regel des Regelwerks kommt sonst eine Objektvariable des Typs Business Activity vor. Folglich kann nur die Umkehrregel von r_{a4} auf die von ihr erzeugten Fragmente matchen.

Regel r_{a5} Objekte des Typs Class Association. Da keine andere Regel Objektvariablen dieses Typs enthält und die beiden Objektvariablen vom Typ Class Association End sich durch den Wert des Attributs „aggregationType“ unterscheiden kann nur die Umkehrregel von r_{a5} genau einmal auf jedes von ihr erzeugte Fragment matchen.

Für den Nachweis von Eigenschaft (iv) aus Definition 4.3.5 muss gezeigt werden, dass für jedes denkbare Quellmodell alle Elemente in mindestens einem Match des Regelwerks enthalten sind.

Business Process Objekte müssen mit mindestens einem Aktor über eine „isPerformedBy“-Assoziation in Beziehung stehen. Folglich werden sämtliche Objekte dieses Typs zumindest von Regel r_{b1} erfasst.

Assoziationen des Typs Business Process isPerformedBy Business Actor werden aus dem oben genannten Grund ebenfalls immer von Regel r_{b1} erfasst.

Assoziationen des Typs Business Process initiated by werden immer von Regel r_{b3} erfasst.

Business Actor Objekte müssen mit mindestens einem Objekt des Typs Business Process über eine „performs“-Assoziation verbunden sein. Folglich werden sämtliche Objekte dieses Typs zumindest von Regel r_{b1} erfasst.

Assoziationen des Typs Business Process uses Business Process werden immer von Regel r_{b4} erfasst.

Objekte des Typs Business Entity müssen immer in einer „usedBy“-Beziehung zu Business Processes stehen. Folglich werden sie immer von Regel r_{b2} erfasst.

Assoziationen des Typs Business Entity contains Business Entity werden immer von Regel r_{b5} erfasst. Der reflexive Fall ist durch die Kompositionsbeziehung ausgeschlossen.

Somit wurde der Nachweis erbracht, dass das Regelwerk R_{BRM} streng match-bijektiv, und somit gemäß Satz 4.3.1 streng bijektiv ist. Folglich spezifiziert R_{BRM} eine gültige Verfeinerung von Business Requirements Modellen bezüglich der in R_a definierten Abstraktion.

Abbildung 5.14 zeigt das Ergebnis der Anwendung des Regelwerks R_{BRM} auf das in Abbildung 5.8 dargestellte Business Requirements Modell. Die Darstellung wurde durch Anwendung der Regeln mit dem in ein Kapitel 6 vorgestellten Werkzeug generiert. Im neu erzeugten Requirements Analysis Modell wurden die Beschreibungen der Aktoren übernommen, für die Geschäftsentität „Order“ wurde eine Klasse als Teil des konzeptuellen Datenmodells mit einer Beschreibung angelegt.

Im nachfolgenden Abschnitt wird diskutiert, wie sich ein so geschaffenes Modell erweitern lässt, indem Informationen aus verschiedenen Entwicklungsartefakten in das Modell integriert werden. Durch

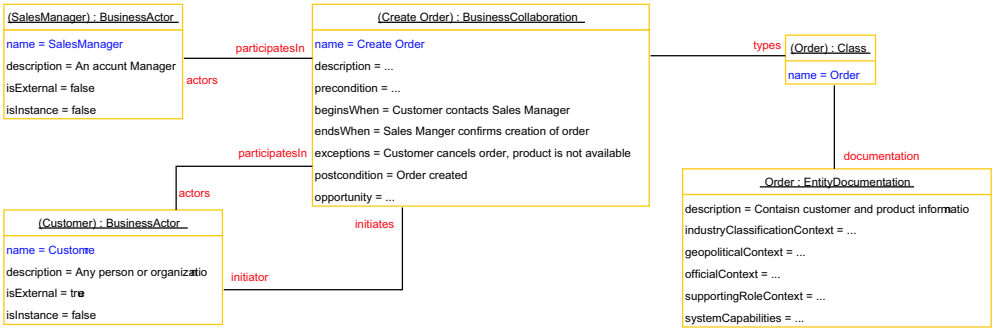


Abbildung 5.14: Das durch die Anwendung des Regelwerks R_{BRM} aus dem in Abbildung 5.8 dargestellten Modell erzeugte Requirements Analysis Modell

Anwendung der durch das Regelwerk R_a spezifizierten Abstraktionsabbildung wird ein so erweitertes Modell wieder auf das ursprüngliche Business Requirements Modell abgebildet, falls durch die Integration zusätzlicher Artefakte die Verfeinerungsbeziehung zwischen den Modellen nicht verletzt wurde.

5.3 Integration von Artefakten in ein konzeptuelles Modell

Für die Modellierung eines Systems ist die direkte Erstellung eines konzeptuellen Modells in der Regel nicht praktikabel. Stattdessen werden im Verlauf eines Entwicklungsprozesses Artefakte geschaffen, durch die jeweils verschiedene Sichten auf ein System beschrieben werden. Hierbei gilt es, die in verschiedenen Artefakten dokumentierten Informationen in ein einheitliches konzeptuelles Modell zu integrieren. Um eine solche Integration zu ermöglichen, wird der Typ eines Artefakts durch die jeweils verwendete Beschreibungstechnik festgelegt (siehe Definition 2.2.6, S. 43). Diese definiert die verwendete konkrete und abstrakte Syntax, sowie die Abbildung der Elemente eines Artefakts auf eine Sicht eines konzeptuellen Modells. Im Rahmen dieses Abschnitts wird dargestellt wie sich BOTL für die Integration von Beschreibungstechniken in ein konzeptuelles Modell einsetzen lässt.

Liegt die abstrakte Syntax einer Beschreibungstechnik in Form eines objektorientierten Klassenmodells vor, so lässt sich die Abbildung ihrer Instanzen in ein objektorientiertes konzeptuelles Modell durch ein BOTL-Regelwerks spezifizieren. Eine solche Abbildung erzeugt jeweils eine Sicht auf das konzeptuelle Modell. Die sich hierbei ergebenden Fragestellungen sind im Wesentlichen:

- Erzeugt die Abbildung der abstrakten Syntax auf ein konzeptuelles Modell korrekte Sichten?
- Können bei der Integration verschiedener Beschreibungstechniken in ein gemeinsames Modell Konflikte auftreten oder sind die verwendeten Beschreibungstechniken orthogonal?

Die Verwendung von BOTL für die Integration von Beschreibungstechniken wird anhand der Integration von Entity-Relationship-Diagrammen und UML-Aktivitätsdiagrammen in ein Requirements Analysis Modell der KOGITO-Methodik exemplarisch dargestellt. Hierzu werden die Regelwerke R_{ER} zur Integration von E/R-Diagrammen und R_{AD} zur Integration von UML-Aktivitätsdiagrammen eingeführt. Abbildung 5.15 skizziert die Erzeugung eines Requirements Analysis Modells durch Verfeinerung eines Business Requirements Modells und die Integration zusätzlicher Artefakte.

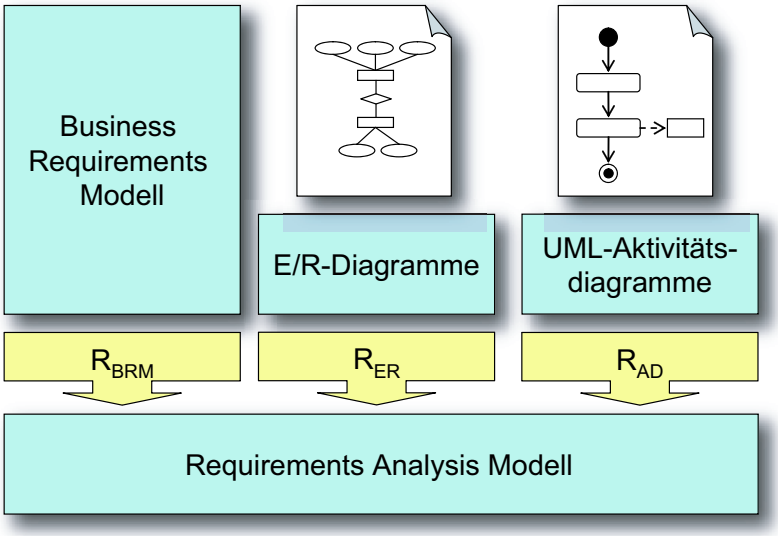


Abbildung 5.15: Integration von Beschreibungstechniken in ein KOGITO Requirements Analysis Modell

5.3.1 Integration von Business E/R-Diagrammen

Innerhalb von KOGITO wird die Beschreibungstechnik „Business E/R-Diagram“ verwendet um konzeptuelle Datenmodelle zu spezifizieren. Die graphische und die abstrakte Syntax der zugrundeliegenden Notation „E/R-Diagramm“ wurde bereits in Abschnitt 5.1 vorgestellt. Das Metamodell mm_{ER} der abstrakten Syntax von ER-Diagrammen ist in Abbildung 5.6 (S. 191) dargestellt.

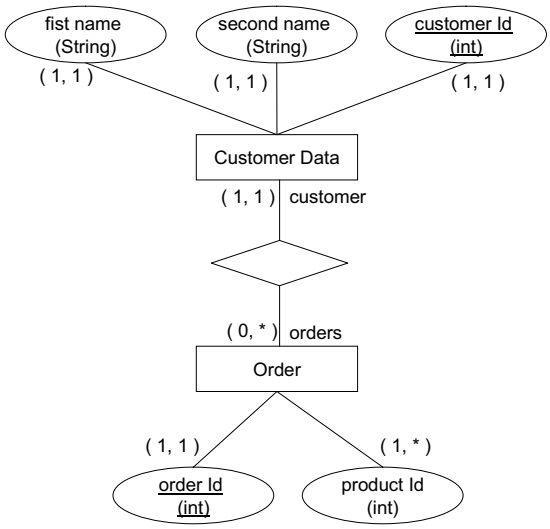


Abbildung 5.16: Beispiel für ein Entity-Relationship-Diagramm

Abbildung 5.16 zeigt ein Beispiel für ein Artefakt des Typs „Business E/R-Diagramm“. Der oben dargestellte Entitätstyp „Customer Data“ verfügt über die drei Attribute „firstName“, „secondName“ und „customerId“, wobei letzteres als Identifikator zur Identifizierung einer Entität dieses Typs dient. Weiterhin existiert eine Beziehung zum Entitätstyp „Order“, die festlegt, dass einer Entität des Typs

„Customer Data“ beliebig viele „Order“-Entitäten zugeordnet werden können. Die Order Entität hat das Attribut „order Id“ als Identifikator und eine Menge aus einem oder mehreren Attributen „product Id“.

Die in KOGITO verwendete Beschreibungstechnik

Business E/R Diagram := (Entity-Relationship-Diagramm, R_{ER})

setzt sich wie in Definition 2.2.6 gefordert aus der Notation „Entity-Relationship-Diagramm“ und der Abbildungsvorschrift R_{ER} auf das konzeptuelle Requirements Analysis Modell zusammen. Durch diese Abbildungsvorschrift wird definiert, wie die Elemente der abstrakten Syntax eines E/R-Diagramms innerhalb des konzeptuellen KOGITO-Modells zu interpretieren sind.

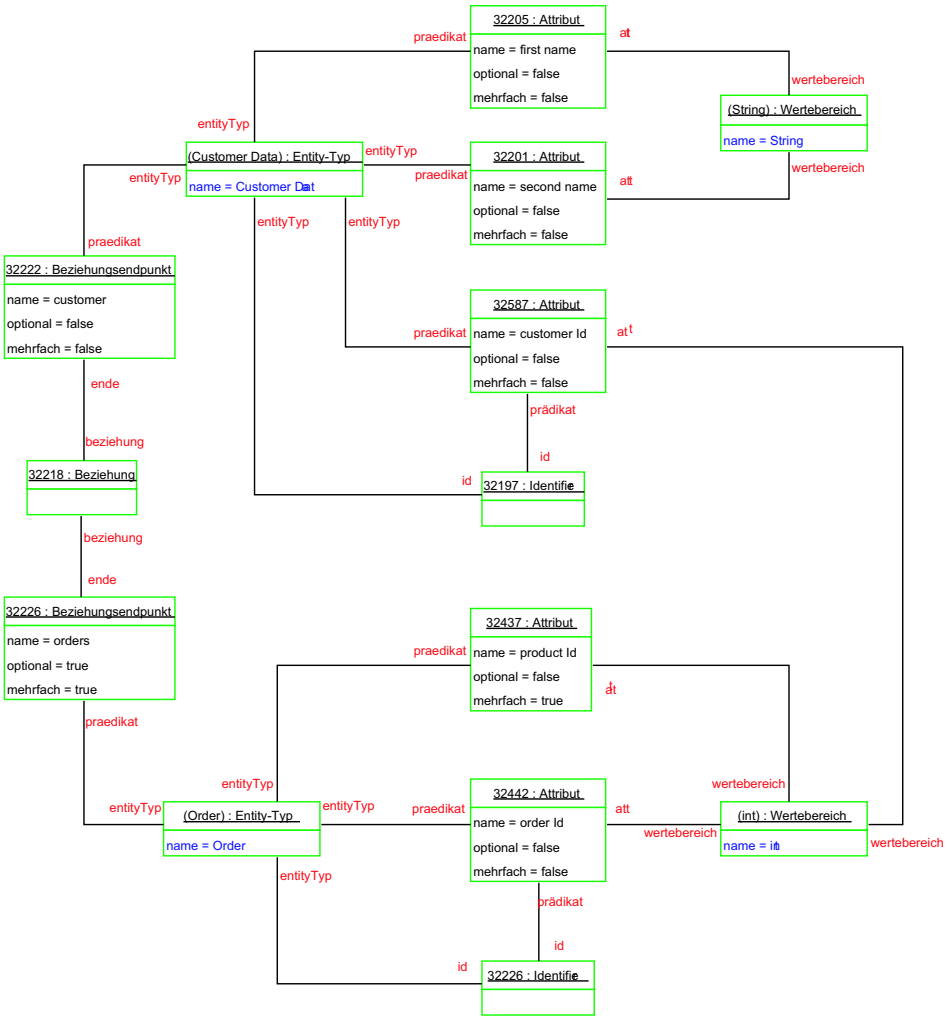


Abbildung 5.17: Instanzmodell m_{ER} der abstrakten Syntax des E/R-Diagramms aus Abbildung 5.16

Um die Abbildung von E/R-Diagrammen auf ein Requirements Analysis Modell exakt spezifizieren zu können, beziehen sich die Abbildungsregeln nicht auf die graphische Darstellung der Diagramme, sondern auf ihre Darstellung als Instanz ihrer abstrakten Syntax. Abbildung 5.17 zeigt das Instanzmodell der abstrakten Syntax m_{ER} für das in Abbildung 5.16 dargestellte E/R-Diagramm.

Das Regelwerk R_{ER} zur Integration von E/R-Diagrammen in ein Requirements Analysis Modell

Die Abbildung der abstrakten Syntax von E/R-Diagrammen auf das KOGITO Requirements Analysis Modell wird durch das Regelwerk R_{ER} spezifiziert. Die Regeln $R_{ER} = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ des Regelwerks sind in den Abbildungen 5.18 bis 5.23 dargestellt. Sie alle haben dasselbe Quellmetamodell mm_{ER} aus Abbildung 5.6. Das Zielmetamodell mm_{RAM} dieses Regelwerks ist das in Abbildung 5.4 dargestellte Metamodell für Requirements Analysis Modelle.

Die einzelnen Objektvariablen der Regeln sind mit Kommentaren versehen, welche jeweils den Namen der Objektvariablen enthalten. Im weiteren Verlauf wird dieser Name auch in einer quantifizierten Schreibweise (z.B. $r_1.cv$ für die Objektvariable *cv* der Regel r_1) verwendet, um Verwechslungen zwischen Objektvariablen verschiedener Regeln auszuschließen. Die von dem Regelwerk erzeugten Modelle enthalten nicht alle Arten von Instanzen des Zielmetamodells mm_{RAM}. Tatsächlich beschränken sie sich auf die zur Modellierung von Klassenmodellen notwendigen Teile des Zielmetamodells, der im folgenden auch als *Business Data Modell* bezeichnet wird.

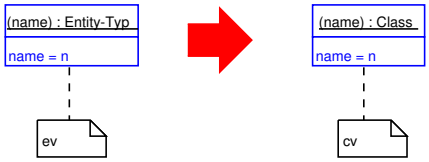


Abbildung 5.18: Abbildungsregel r_1

Regel r_1 erzeugt für jeden Entitätstyp eines E/R-Diagramms eine Klasse im Business Data Model des Requirements Analysis Modells. In beiden Klassen dient der Name als Primärschlüssel. Durch diese Regel wird sichergestellt, dass auch Entitätstypen, die über keinerlei Assoziationen oder Attribute verfügen als Klasse im Business Data Model enthalten sind.

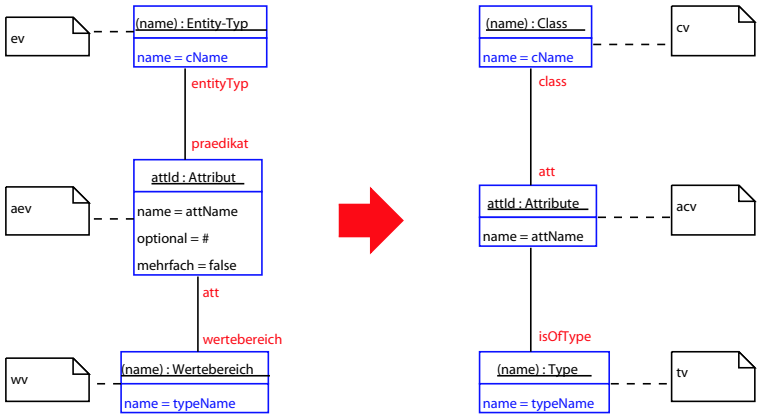


Abbildung 5.19: Abbildungsregel r_2

Die Regel r_2 stellt sicher, dass für jedes Attribut eines Entitätstyps, das nicht mehrfach vorkommen darf genau ein Attribut in der entsprechenden Klasse im RAM existiert. Die Bezeichnung des Wertebereichs des Attributs im E/R-Diagramm entspricht hierbei dem Typ des Attributs innerhalb der Klasse. Im Gegensatz zur UML werden Attribute mit einer größeren Multiplizität als eins (z.B. Arrays) nicht vom Requirements Analysis Modell unterstützt. Für solche Attribute erfolgt eine gesonderte Behandlung in den Regeln r_3 und r_4.

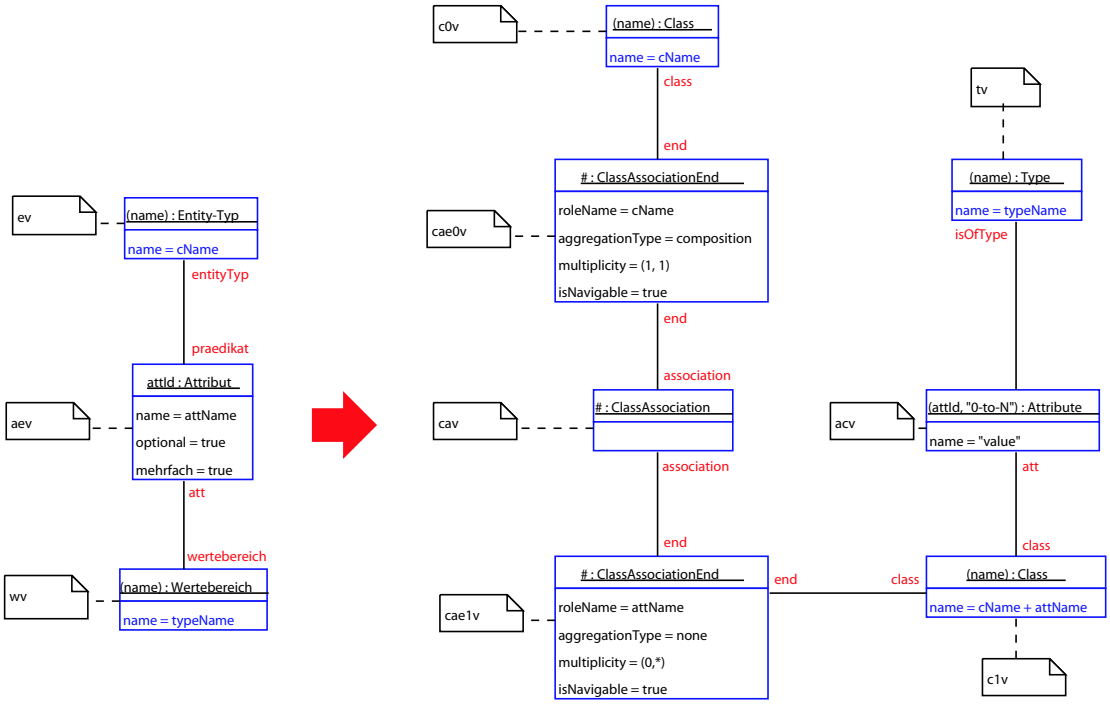


Abbildung 5.20: Abbildungsregel r_3

Durch die Regel r_3 wird für jedes Attribut eines Entitätstyps mit einer Multiplizität „0..*“ eine eigenes Class-Objekt im Zielmodell erzeugt. Dieses ist über eine Kompositionsbeziehung mit mit dem Class-Objekt, welches die Entität repräsentiert verbunden. Abbildung 5.21 veranschaulicht die Auswirkung der Anwendung von Regel r_3 exemplarisch. Die obere Hälfte zeigt eine Entität mit beliebig vielen Attributen „att“ des Typs int. Im unteren Bildabschnitt ist das aus diesem Diagramm erzeugte KOGITO-Datenmodell als Klassendiagramm dargestellt. Für das Attribut „att“ wurde eine eigene Container-Klasse erzeugt, die beliebig oft in Instanzen der Klasse ACLass enthalten sein kann.

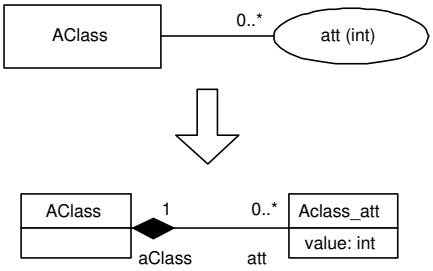


Abbildung 5.21: Beispiel für eine Anwendung der Regel r_3

Für Attribute mit einer Multiplizität von „1..*’"legt Regel r_4 vollkommen analog zu Regel zwei über eine Kompositionsbeziehung verbundene Klassen im Zielmodell an.

Regel r_5 bildet die Identifikatoren von Entitätstypen auf Primärschlüssel von Klassen im Business Data Modell ab. Da das Zielmetamodell keine Attribute mit einer Multiplizität größer als 1 in Klassen unterstützt werden hier lediglich die Identifikatoren auf Primärschlüssel abgebildet, die sich auf einfache Attribute eines Entitätstyps beziehen.

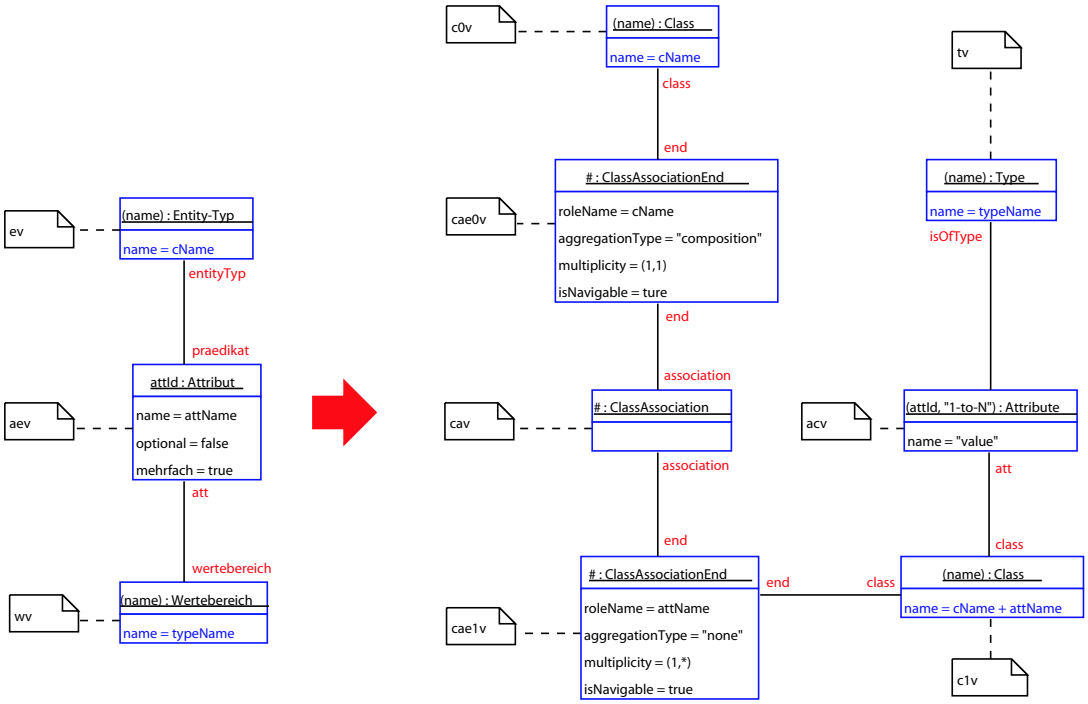


Abbildung 5.22: Abbildungsregel r_4

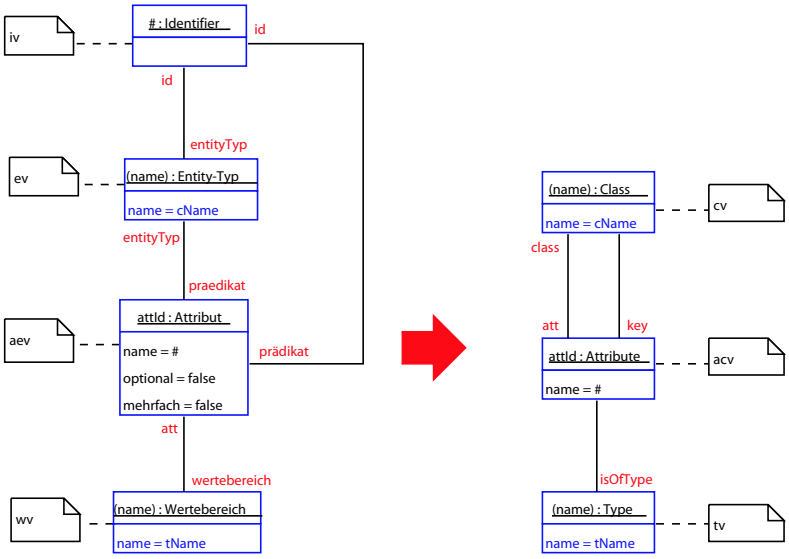


Abbildung 5.23: Abbildungsregel r_5

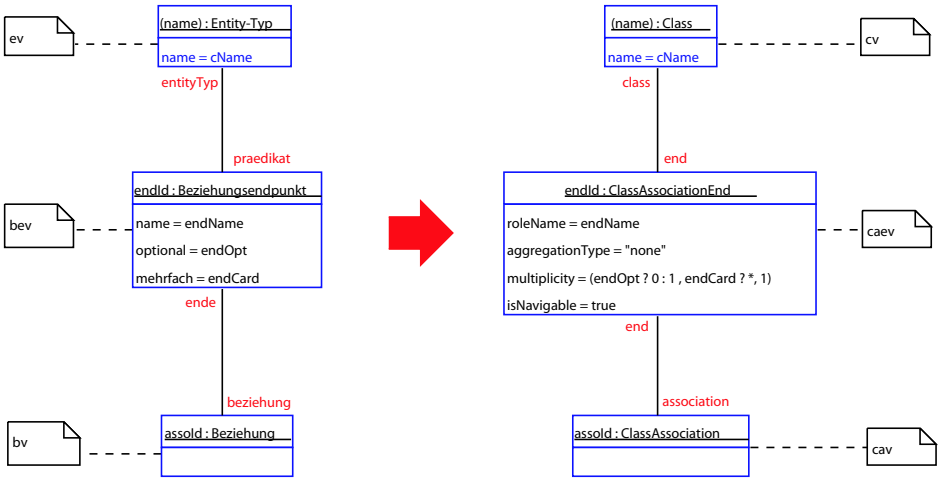


Abbildung 5.24: Abbildungsregel r_6

Beziehungen zwischen zwei Entitätstypen werden durch Regel r_6 auf Assoziationen zwischen Klassen abgebildet. Die vorliegende Regel erzeugt jeweils nur eine „halbe“ Assoziation. Sie muss folglich für jede Beziehung im E/R-Modell zweimal angewendet werden. In den Abschnitten B.2 und B.3 wird gezeigt, dass sie dennoch nur Assoziationen mit genau zwei Assoziationsenden erzeugt.

Anstelle mehrerer Regeln für verschiedene Assoziationsmultiplizitäten zu verwenden, wie dies im Falle von Regel r_3 und r_4 getan wurde, wird hier der Ausdruck $endOpt?0 : 1$ bzw. $endCard?* : 1$ in der Zielmodellvariable verwendet. Dieser Term liefert den Wert der Multiplizitätsobergrenze bzw. -untergrenze in Abhängigkeit von dem Boolschen Wert $endOpt$ bzw. $endCard$. Die Verwendung dieser Notation erspart oftmals das schreiben weitgehend redundanter Regeln, wobei jedoch beide Lösungen zu metamodellkonformen Zielmodellen führen sind, wie in Abschnitt B gezeigt wird.

Erzeugen von Sichten aus E/R-Diagrammen

Die durch das Regelwerk R_{ER} definierte Transformation liefert zu einem Instanzmodell der abstrakten Syntax eine Sicht auf das Requirements Analysis Modell. Die in diesem Fall erzeugte Sicht „Business Data Model“ beschreibt die Struktur von verarbeiteten Informationen des Systems.

Das Ergebnis der Transformation des Modells der abstrakten Syntax aus Abbildung 5.17 durch das Regelwerk R_{ER} ist in Abbildung 5.25 dargestellt. Die beiden Objektdiagramme wurden mit Demonstrationsmodus des in Abschnitt 6 vorgestellten Werkzeuges für BOTL-Transformationen erstellt.

Das innerhalb von KOGITO verwendet Business Data Metamodell als Teil des Business Requirements Metamodells entspricht genau dem in BOTL verwendeten Metamodell für die Spezifikation von Klassenmodellen (siehe auch Abbildung 6.3 auf Seite 236). Dementsprechend lassen sich seine Instanzen in Form eines für den Anwender leichter lesbaren Klassendiagramms darstellen. Abbildung 5.26 zeigt exemplarisch die Darstellung des Business Data Modells aus Abbildung 5.25 in Form eines solchen Klassendiagramms. Wie sich leicht erkennen lässt, wurden Attribute von Entitäten zu Attributen der jeweils aus ihnen erzeugten Klassen. Lediglich für das Attribut „product Id“ wurde eine eigene Container-Klasse erzeugt, da es als einziges mehrfach auftreten kann.

Um sicherstellen zu können, dass die durch die Anwendung des Regelwerks R_{ER} keine inkonsistenten Sichten erzeugt werden, kann die Metamodellkonformität des Regelwerks verifiziert werden. Dar Nachweis der Metamodellkonformität dieses Regelwerks findet sich in Anhang B.

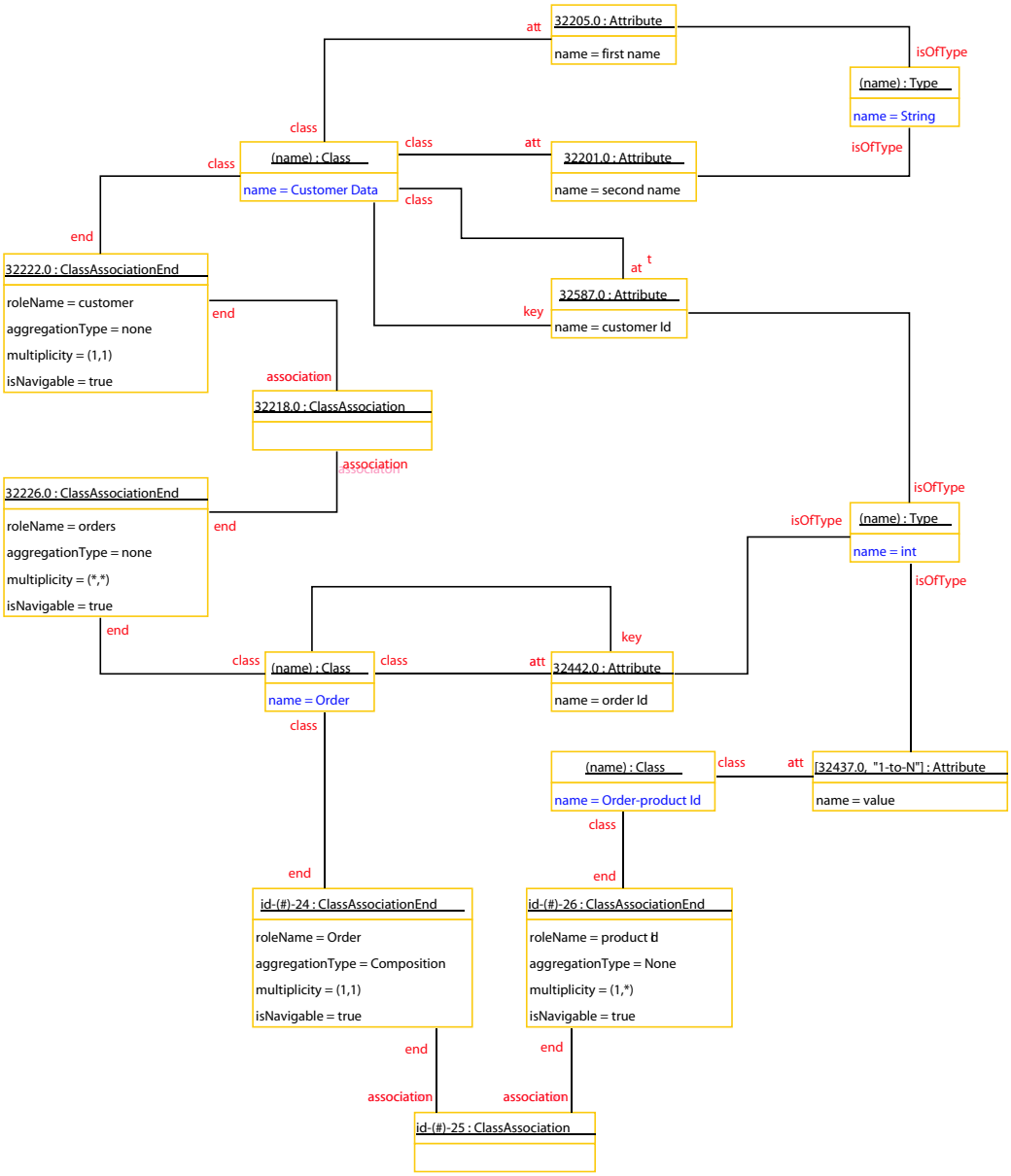


Abbildung 5.25: Die aus dem E/R-Diagramm erzeugte Sicht des konzeptuellen KOGITO-Modells

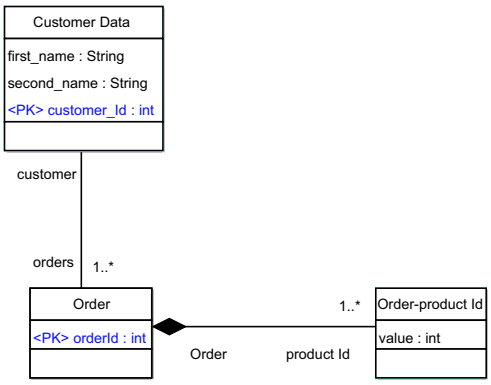


Abbildung 5.26: Darstellung des konzeptuellen Datenmodells aus Abbildung 5.25 in Form eines Klassendiagramms

5.3.2 Integration mehrerer Beschreibungstechniken

Im Verlauf einer Systementwicklung werden in der Regel verschiedene Beschreibungstechniken zur Erstellung und Manipulation unterschiedlicher Sichten auf ein konzeptuelles Modell verwendet. So werden die im vorangegangenen Abschnitt vorgestellten Business E/R-Diagramme verwendet, um Daten, die von einem System verarbeitet werden, zu modellieren. Für andere Aspekte einer Systembeschreibung werden jeweils weitere, angepasste Beschreibungstechniken benötigt.

Für die Umsetzung eines modellbasierten Entwicklungsansatzes ist es hierbei wichtig zu wissen, ob die Integration der unterschiedlichen Beschreibungstechniken in ein gemeinsames konzeptuelles Modell zu Inkonsistenzen führen kann und wodurch diese ggf. verursacht werden können. Sind mögliche Konflikte zwischen Beschreibungstechniken im Vornherein bekannt, so können diese gezielt durch eine Werkzeugunterstützung behandelt werden.

Business Activity Diagramme

Innerhalb von KOGITO wird die Beschreibungstechnik

Business Activity Diagramm := (UML Aktktivitätsdiagramm, R_{AD})

verwendet, um den Ablauf von Geschaftsprozessen zu modellieren. Artefakte dieses Typs verwenden als Notation ein Profil für UML-Aktiviätsdiagramme, d.h. die abstrakte Syntax dieser Notation ist durch das UML Metamodell festgelegt. Die für das hier angeführte Beispiel relevanten Regeln des Regelwerks R_{AD} finden sich in Anhang C (S. 339).

Der Vorteil der Verwendung von UML-Notationen für eigene Beschreibungstechniken liegt in der Möglichkeit Standard-UML-Werkzeuge für die Erstellung von Artefakten zu verwenden. Soll eine Methodik wie KOGITO durch ein Werkzeug mit einem zentralen Repository unterstützt werden, so können die UML-Artefakte durch das BOTL-Werkzeug (siehe Kapitel 6) automatisiert in ein zentrales KOGITO-Modell integriert werden.

Abbildung 5.27 zeigt ein Beispiel für ein KOGITO-Artefakt des Typs „Business Activity Diagram“, welches mit dem UML-Werkzeug Poseidon [AG04] erstellt wurde. Das Aktivitätsdiagramm beschreibt zwei, zur Realisierung des Geschäftsprozesses „Create Order“ notwendige Aktivitäten.

Abbildung 5.28 stellt das Instanzmodell m_{AD} der abstrakten Syntax des Aktivitätsdiagramms dar, wie es durch das Modellierungswerkzeug Poseidon erzeugt wird. An dem Beispiel wird ersichtlich, dass UML-Diagramme zu vergleichsweise komplexen internen Modellen führen. Dies erschwert die

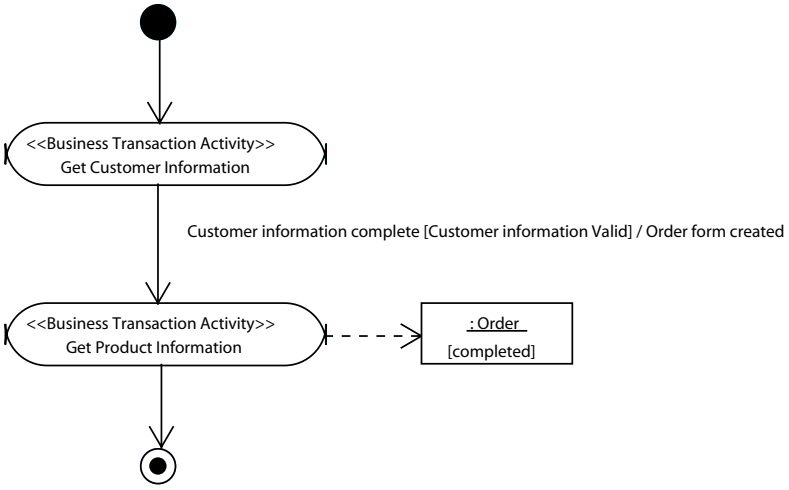


Abbildung 5.27: Das Artefakt „Create Order“ vom Typs „Business Activity Diagram“

Verwendung des UML-Metamodells als konzeptuelles Metamodell für die Beschreibung von Systemen innerhalb von eng abgegrenzten Anwendungsdomänen. Es bietet sich daher an, die Instanzmodelle der abstrakten Syntax in ein domänenspezifisches konzeptuelles Modell, wie das Requirements Analysis Modell von KOGITO, zur transformieren.

Das Ergebnis der Transformation des Modells aus Abbildung 5.28 ist in Abbildung 5.29 dargestellt. Das so erzeugte Modellfragment ist für sich alleine jedoch nicht konform zum Metamodell mm_{RAM} sondern lediglich zu dem Ausschnitt des Metamodells der für die Spezifikation des Verhaltens zuständig ist. Dementsprechend stellt dieses Modellfragment eine Sicht auf das gesamte konzeptuelle Modell dar.

Integration von Beschreibungstechniken

Grundsätzliches Um mehrere Beschreibungstechniken in ein konzeptuelles Modell integrieren zu können, muss gemäß Definition 2.2.7 (S. 45) ein *Kompositionsoperator* zur Komposition der durch die einzelnen Beschreibungstechniken erzeugten Sichten existieren. Im hier vorgestellten Ansatz wird der in Definition 3.5.12 (S. 95) vorgestellte $\cup_{\mathfrak{m}}$-Operator für die Komposition objektorientierter Modelle verwendet.

Dementsprechend ergibt sich ein konzeptuelles Modell als Komposition der durch die Artefakte der Beschreibungstechniken gebildeten Sichten. Sei mm_{cm} das Metamodell des konzeptuellen Modells und $B = \{b_0, \ldots, b_n\}$ eine Menge von Beschreibungstechniken deren jeweilige Transformationsspezifikation in Form eines BOTL-Regelwerks definiert wurde, so dass gilt:

$$\forall b \in B : b|_{mt} = R \in RW \wedge mm_R^1 = mm_{cm}$$

D.h. für alle Beschreibungstechniken existiert eine Abbildungsvorschrift der abstrakten Syntax in das selbe konzeptuelle Modell. Für eine Menge von zugehörigen Instanzmodellen der abstrakten Syntax $\{m_0, \ldots, m_n\}$ ergibt sich somit ein konzeptuelles Modell m_k als Ergebnis der folgenden *Integrationstransformation*:

$$\begin{aligned} m_k &:= int[b_0, \ldots, b_n](m_0, \ldots, m_n) \\ &\overset{\text{Def. 2.2.8 (S. 46)}}{=} transform(m_0, b_0|_{mt}) \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} transform(m_n, b_n|_{mt}) \end{aligned}$$

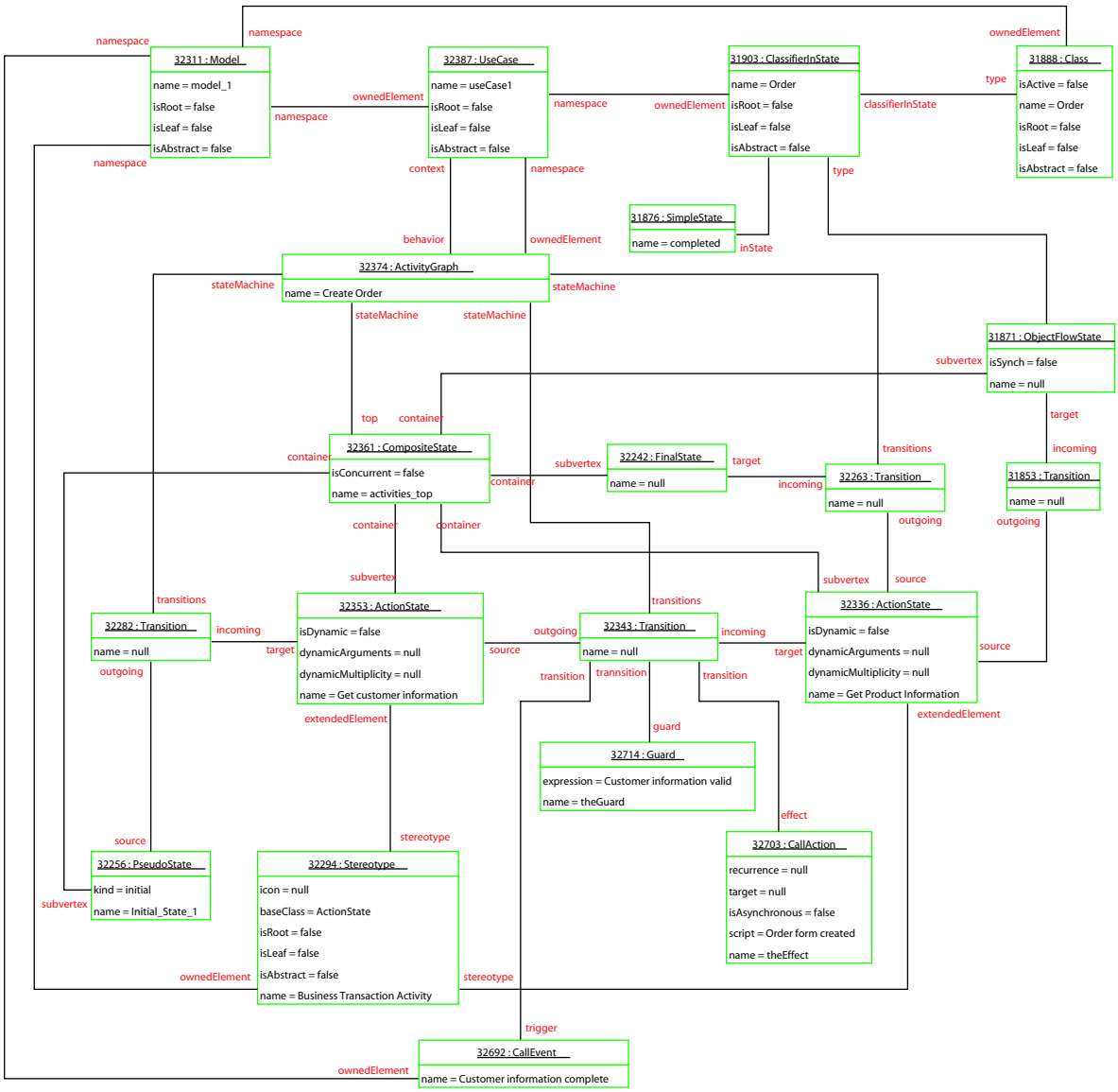


Abbildung 5.28: Instanzmodell m_{AD} des in Abbildung 5.27 dargestellten Artefakts

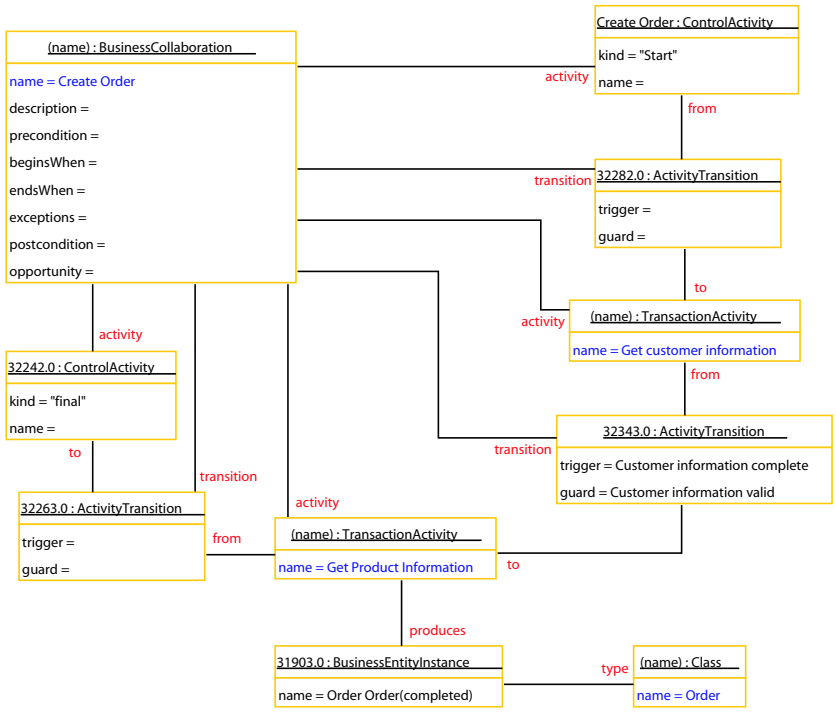


Abbildung 5.29: Die aus dem Aktivitätsdiagramm „Create Order“ erzeugte Sicht des konzeptuellen KOGITO-Modells

Wird das konzeptuelle Modell nicht vollständig durch eine Menge von Artefakten aufgebaut, sondern entsteht zu Teilen durch eine Verfeinerung eines abstrakteren Ursprungsmodells, kann auch dieser Sachverhalt berücksichtigt werden, indem diese Verfeinerungsabbildung in die Integration der Sichten mit einbezogen wird.

Erzeugen eines konzeptuellen Modells So entsteht das Requirements Analysis Modell im hier behandelten Beispiel durch die Komposition des verfeinerten Business Requirements Modells mit den jeweils erzeugten Sichten:

$$\begin{aligned} m_{RAM} := \quad & transform(\{m_{BRM}\}, R_{BRM}) \\ & \cup_{\mathfrak{m}} transform(\{m_{ER}\}, R_{ER}) \\ & \cup_{\mathfrak{m}} transform(\{m_{AD}\}, R_{AD}) \end{aligned}$$

Das Ergebnis der Komposition dieser drei erzeugten Modellfragmente ist in Abbildung 5.30 dargestellt. Interessant ist es nun Aussagen darüber zu machen, ob ein solches Zusammenführen verschiedener Artefakte in ein bestehendes Modell ohne Konflikte möglich ist. Gemäß Satz 3.5.6 (siehe S. 101) gilt allgemeinen für eine Menge von Regelwerken $\{R_0, \ldots R_n\} \in \mathscr{P}(\mathbb{RW})$ mit jeweils unterschiedlichen Quellmetamodellen und den jeweiligen Quellmodellen $\{m_0\}, \ldots, \{m_n\}$:

$$transform(m_0, R_0) \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} transform(m_n, R_n) = transform(\{m_0, \ldots, m_n\}, R_0 \cup \cdots \cup R_n)$$

Dementsprechend gilt für das hier behandelte Beispiel:

$$m_{RAM} = transform\big(\{m_{BRM}, m_{ER}, m_{AD}\}, R_{BRM} \cup R_{ER} \cup R_{AD}\big)$$

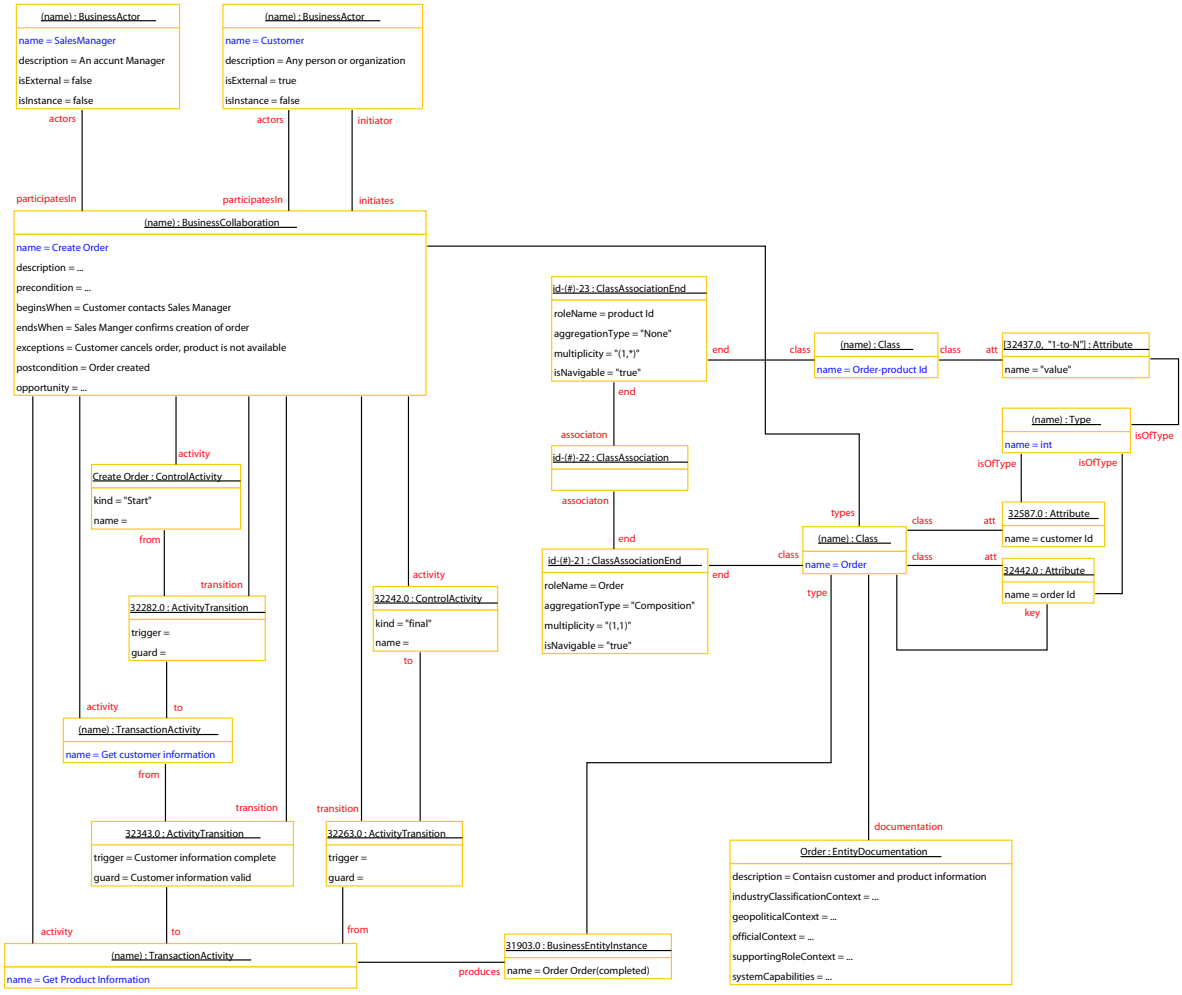


Abbildung 5.30: Das durch Zusammenführen der erzeugten Modellfragmente entstandene Requirements Analysis Modell m_{RAM}

Orthogonalität der Sichten Um Nachzuweisen, dass die beiden verwendeten Beschreibungstechniken orthogonal sind, muss gemäß Definition 2.2.9 gelten:

$$\forall m_{ER} \in \mathbb{M}_{mm_{ER}}, m_{AD} \in \mathbb{M}_{mm_{AD}} : transform\big(\{m_{ER}, m_{AD}\}, R_{ER} \cup R_{AD}\big) \in \mathbb{M}_{mm_{RAM}}$$

Allerdings wird das RAM nicht allein durch die Integration der Artefakte aus diesen Beschreibungstechniken gebildet. Stattdessen wird im vorliegenden Fall bestehendes RAM durch die Integration der Artefakte erweitert. Das Ergebnis dieser Erweiterung eines bestehenden RAM lässt sich als Anwendung des Rewrite-Regelwerks

$$RW_{int} := R_{ER} \cup R_{AD} \in \mathbb{RRW} \text{ mit } \forall r \in RW_{int} : tx(RW, i) = 0$$

darstellen. Dieses Rewrite-Regelwerk verfügt offensichtlich über keine Rewrite-Regeln. Es unterscheidet sich jedoch von einem gewöhnlichen Regelwerk dadurch, dass bei seiner Anwendung bereits ein beliebiges, zu mm_{RAM} konformes Zielmodell existieren kann. Für das Ergebnis der Anwendung des Regelwerks gilt dementsprechend für zwei Quellmodelle m_{ER}, m_{AD} und ein gegebenes Requirements Analysis Modell m_{RAM}:

$$m'_{RAM} = rwTransform(\{m_{RAM}, m_{ER}, m_{AD}\}, R_{ER} \cup R_{AD}, (0))$$

Folglich muss gemäß Definition 2.2.9 für den Nachweis der Orthogonalität aller dieser Sichten gezeigt werden, dass gilt:

$$\forall m_{ER} \in \mathbb{M}_{mm_{ER}}, m_{AD} \in \mathbb{M}_{mm_{AD}} m_{RAM} \in \mathbb{M}_{mm_{RAM}} : \\ rwTransform\big(\{m_{RAM}, m_{ER}, m_{AD}\}, RW_{int}\big) \in \mathbb{M}_{mm_{RAM}}$$

Hierzu gilt es gemäß Definition 4.4.1 (S. 171) die Metamodellkonformität dieses Rewrite-Regelwerks nachzuweisen. Gemäß Satz 4.4.2 muss gelten:

$$mmConform(R^{id}_{mm_{RAM}} \cup RW^0_{int}) = mmConform(R^{id}_{mm_{RAM}} \cup R_{ER} \cup R_{AD})$$

Ein solcher Nachweis ist aufgrund der vergleichsweise schwachen Verifikationstechniken für Rewrite-Regelwerke gegenüber denen für normale Regelwerke *nicht* möglich. Dieser Sachverhalt ist leicht nachvollziehbar, da die Orthogonalität der Sichten für *beliebige* Requirements Analysis Modells, in welche Artefakte integriert werden, eine sehr weitreichende Forderung ist.

Tatsächlich ist jedoch bekannt, dass sämtliche ursprünglichen Requirements Analysis Modelle durch die Verfeinerung eines Business Requirements Modells erzeugt wurden. Wird diese zusätzliche Einschränkung in Betracht gezogen, so muss für den Nachweis der Orthogonalität der Sichten gezeigt werden, dass gilt:

$$\forall m_{BRM} \in \mathbb{M}_{mm_{BRM}}, m_{ER} \in \mathbb{M}_{mm_{ER}}, m_{AD} \in \mathbb{M}_{mm_{AD}} : \\ transform\big(\{m_{BRM}, m_{ER}, m_{AD}\}, R_{BRM} \cup R_{ER} \cup R_{AD}\big) \in \mathbb{M}_{mm_{RAM}}$$

die Einschränkung, dass das Requirements Analysis Modell durch eine Verfeinerungsabbildung erzeugt wurde, schränkt die Menge der möglichen „Quellmodelle“ dieses Typs weiter gegenüber der zuvor vorgestellten Lösung weiter ein. Wie sich zeigen lässt ist das Regelwerk $R_{BRM} \cup R_{ER} \cup R_{AD}$ metamodellkonform, auf einen detaillierten Nachweis wird an dieser Stelle jedoch aus Platzgründen verzichtet. Das zusätzliche Wissen über die möglichen Requirements Analysis Modelle erlaubt somit den Nachweis der Orthogonalität der Sichten für einer größere Klasse von Beschreibungstechniken.

Es kann also gezeigt werden, dass durch die Abbildung der Beschreibungstechniken „Business Entity Diagram“ und „Business Activity Diagram“ in ein verfeinertes Requirements Analysis Modell niemals Inkonsistenzen in diesem Modell auftreten können. Die beiden Beschreibungstechniken sind dementsprechend *orthogonal* gemäß Definition 2.2.9 (S. 46).

Generell lässt sich für eine Methodik in den meisten Fällen die Orthogonalität aller verwendete Beschreibungstechniken nicht nachweisen, da oftmals Beschreibungstechniken, wie z.B. Automaten und Sequenzdiagramme, eingesetzt werden, welche zu inkonsistenten Modellen führen können. Die Verwendung solcher nicht orthogonaler Beschreibungstechniken ist in der Praxis jedoch gerechtfertigt, da sie oftmals die Spezifikation von Systemen erleichtern. So werden beispielsweise Automaten verwendet um das Verhalten einer Komponente zu beschreiben, während gewünschte Abläufe anhand von Sequenzdiagrammen dokumentiert werden. Im Verlauf des Entwicklungsprozesses gilt es schließlich sicherzustellen, dass das durch den Automaten spezifizierte Verhalten einer Komponente alle durch Sequenzdiagramme modellierten Abläufe unterstützt. Ziel eines Entwicklungsprozesses ist es somit sicherzustellen, dass auch die Artefakte nicht orthogonaler Beschreibungstechniken zu einem konsistenten Modell führen.

Die Überprüfung der Metamodellkonformität der durch die Integration von Beschreibungstechniken entstehenden Abbildung ist dennoch sinnvoll, da sie es erlaubt bereits bei der Definition der Abbildungsvorschriften potentielle Konflikte von Artefakten des jeweiligen Typs zu identifizieren. Diese können dann gezielt durch eine Werkzeugunterstützung erkannt werden, bzw. durch ein angepasstes Vorgehen weitestgehend vermieden werden.

Verfeinerungskomposition und konsistente Erweiterung Damit die Integration der verschiedenen Sichten eine *Verfeinerungskomposition* bezüglich der Abstraktionsabbildung R_A darstellt, muss gemäß Definition 2.2.10 (S. 46) gelten:

$$\begin{aligned}&\forall m_{BRM} \in \mathbb{M}_{mm_{BRM}}, m_{ER} \in \mathbb{M}_{mm_{ER}}, m_{AD} \in \mathbb{M}_{mm_{AD}} : \\ &\quad m_{BRM} = transform\big(transform(\{m_{BRM}, m_{ER}, m_{AD}\}, R_{BRM} \cup R_{ER} \cup R_{AD}), R_a\big)\end{aligned}$$

Das dies für das gegebene Beispiel *nicht* der Fall ist lässt sich leicht durch die Angabe eines Gegenbeispiels zeigen. Ein solches Beispiel findet sich in Anhang D auf Seite 345 ff.

Allerdings lässt sich nachweisen, dass die durch das Regelwerk $R_{BRM} \cup R_{ER} \cup R_{AD}$ spezifizierte Integrationstransformation eine *konsistente Erweiterung* bezüglich der Abstraktionsabbildung R_a darstellt (siehe Definition 2.2.12, S. 48). als Teilmodelloperator wird hierbei der $\dot{\subseteq}$-Operator aus Definition 3.5.13 angenommen.

Generell gilt für Mengen von beliebigen aber festen Quellmodellen $M = \{m_{BRM}, m_{ER}, m_{AD}\}$ wie sie hier vorliegen:

$$\begin{aligned}
& m_{BRM} \\
\dot{\subseteq}\ & \underbrace{transform\big(transform(M, R_{BRM}), R_a\big)}_{=m_{BRM} \text{ (siehe Abschnitt 5.2)}} \cup_{\mathfrak{m}} transform\big(transform(M, R_{ER}), R_a\big) \cup_{\mathfrak{m}} \\
& transform\big(transform(M, R_{AD}), R_a\big) \\
\overset{\text{Satz 3.5.7}}{\dot{\subseteq}}\ & transform\big(transform(M, R_{BRM}) \cup_{\mathfrak{m}} transform(M, R_{ER}) \cup_{\mathfrak{m}} transform(M, R_{AD}), R_a\big) \\
\overset{\text{Satz 3.5.6}}{=}\ & transform\big(transform(M, R_{BRM} \cup R_{ER} \cup R_{AD}), R_a\big)
\end{aligned}$$

Somit ist die hier vorgestellte Integrationsabbildung eine konsistente Erweiterung gemäß Definition 2.2.12. D.h. alle Aussagen, die innerhalb eines BRM gemacht werden haben auch innerhalb des eines RAM Gültigkeit, sofern dieses durch eine Verfeinerung des BRM und die Integration zusätzlicher Artefakte des Typs „Business E/R Diagram“ oder „Business Activity Diagram“ erzeugt wurde. Allerdings kann das RAM durch die Integration zusätzlicher Artefakte in einer Weise erweitert werden, in der es keine gültige Verfeinerung des ursprünglichen BRM mehr darstellt, z.B. indem Aktoren eingeführt werden, die innerhalb des BRM nicht bekannt sind. In einem solchen Fall muss das BRM entsprechend erweitert werden, bzw. durch eine Abstraktionsabbildung neu erzeugt werden.

5.4 BOTL-Transformation zur Spezifikation von Vorgehensschritten

Die Untersuchung verschiedener bestehender Software-Entwicklungsprozesse, wie z.B. dem Rational Unified Process (RUP) [Kru00a] innerhalb des Forschungsprojektes ZEN [BBE+03] hat gezeigt, dass das beschriebene Vorgehen oftmals Inkonsistenzen aufweist. So wird beispielsweise im Verlauf eines Entwicklungsprozesses auf Artefakte Bezug genommen, die zu diesem Zeitpunkt noch nicht existieren oder aber das Vorgehen ist lückenhaft dokumentiert beschreibt nicht für alle Entwicklungsartefakte die notwendigen Schritte zu deren Erzeugung. Gerade wenn ein Entwicklungsprozess durch die Auswahl einer Menge adäquater Vorgehensbausteine speziell für ein Projekt zugeschnitten wird, sind solche Inkonsistenzen häufig vorzufinden.

Um solche Inkonsistenzen zu vermeiden werden in dem in Abschnitt 2.2.2 vorgestellten Prozessmusteransatz Vorgehensschritte explizit als Transformationen des Modells der Entwicklungsartefakte dokumentiert. Um diese Transformationen präzise definieren zu können, wird jedoch eine geeignete Sprache benötigt. Zudem ist es wünschenswert für eine Menge ausgewählter Vorgehensbausteine formal und automatisiert nachweisen zu können, dass durch sie ein realisierbarer Prozess definiert wird. Dies bedeutet im Einzelnen:

- Es muss überprüfbar sein, ob ein Prozessmuster das eine Aktivität realisiert mit den verfügbaren Eingaben alle erforderlichen Ausgaben produziert.
- Wird ein Prozessmuster durch die Ausführung einer Reihe von feingranulareren Prozessmustern realisiert, so muss sichergestellt werden können, dass deren Ausführung den gewünschten Ergebniskontext produziert.
- Das Modell der Artefakte ist zu jedem Zeitpunkt der Entwicklung konsistent zu dem gewählten Produktmodell (siehe Definition 2.1.6, S. 26).

Da diese Möglichkeiten im bestehenden Prozessmusteransatz nicht existieren werden im Folgenden die in Abschnitt 3.6 vorgestellten BOTL-Rewrite-Transformationen zur Spezifikation des Kontextes von Prozessmustern und Aktivitäten eingeführt. Durch der Möglichkeit die Metamodellkonformität von Rewrite-Regelwerken formal zu verifizieren, lassen sich die erforderlichen Nachweise für die Korrektheit eines so spezifizierten Prozessmodells erbringen.

5.4.1 Spezifikation von Vorgehensschritten

Prozessmuster und Aktivitäten beschreiben Transformationen eines Modells der Artefakte auf verschiedenen Granularitätsebenen. So kann ein Prozessmuster die Erzeugung eines einzelnen Artefakts oder eine vollständige Entwicklungsphase dokumentieren. Dementsprechend werden Prozessmustern und Aktivitäten in *elementare* und *komplexe* Vorgehensschritte unterteilt:

Elementare Vorgehensschritte spezifizieren die Erzeugung oder Manipulation eines Fragments des Modells der Artefakte. Für einen gegebenen Eingabekontext liefern sie exakt einen Ausgabekontext aus Entwicklungsartefakten und Beziehungen zwischen diesen. Ein elementarer Vorgehensschritt wird durch eine Menge von BOTL-Rewrite-Transformationen mit demselben, eindeutigen *tx*-Wert spezifiziert.

Komplexe Vorgehensschritte können als Ausgabekontext eine Menge möglicher Artefakte und Beziehungen zwischen ihnen haben. Ihr Ausgabekontext wird durch die Angabe eines Metamodells spezifiziert, dass die Struktur der durch den Vorgehensschritt erzeugbaren Artefaktmodelle definiert.

Elementare Prozessmuster referenzieren keine Unteraktivitäten. Statt dessen spezifizieren sie die Transformation des Artefaktmodells in Form eines Rewrite-Regelwerks. Hierbei müssen sämtliche Regeln denselben *tx*-Wert aufweisen. Dementsprechend wird ein elementares Prozessmuster in Form einer „Transaktion“ ausgeführt.

Definition 5.4.1 (Elementares Prozessmuster)
Ein *elementares Prozessmuster* $p = (id, R)$ ist ein Tupel aus

- einem eindeutigen Identifikator $id \in \mathbb{ID}$ und
- eine Transformationsspezifikation $R \in RRW$ mit $\forall r_i, r_j \in R : tx(R, r_i) = tx(R, r_j)$. ❍

Analog zu elementaren Prozessmustern werden nun elementare Aktivitäten definiert. Formal sind elementare Aktivitäten und Prozessmuster zwar gleich, in der Praxis ist jedoch die eigentliche Beschreibung eines Vorgehens in Form eines für den Entwickler verständlichen Textes das wesentliche Merkmal elementarer Prozessmuster. Da diese Beschreibungen jedoch für die Überprüfung von Konsistenzkriterien eines Prozessmodells nicht relevant sind wurden sie an dieser Stelle nicht weiter berücksichtigt.

Definition 5.4.2 (Elementare Aktivität)
Eine *elementare Aktivität* $a = (id, R)$ ist ein Tupel aus

- einem eindeutigen Identifikator $id \in \mathbb{ID}$,
- eine Transformationsspezifikation $R \in RRW$ mit $\forall r_i, r_j \in R : tx(R, r_i) = tx(R, r_j)$. ❍

Komplexe Vorgehensschritte sind typischer Weise Prozessmuster die für die Ausführung einer Aktivität auf eine Reihe untergeordneter Aktivitäten verweisen. Definition 5.4.3 gibt eine formale Definition für komplexe Prozessmuster an. Auf die textuellen Anteile von Prozessmustern wird in der Definition verzichtet, da sie für den formalen Nachweis von Konsistenzeigenschaften eines Prozessmodells unerheblich sind.

Definition 5.4.3 (Komplexes Prozessmuster)
Ein *komplexes Prozessmuster* $p = (id, mm_i, mm_r, \mathbb{RA})$ ist ein Tupel aus

- einem eindeutigen Identifikator id,
- Initialmetamodell $mm_i \in \mathbb{MM}$,
- Ergebnismetamodell $mm_r \in \mathbb{MM}$ und

- einer Menge

$$\mathbb{RA} = \{RA^1, \ldots, RA^n\}$$

aus Mengen von referenzierter Aktivitäten, wobei für die Ausführung des Prozessmusters aus jeder dieser Mengen eine Aktivität ausgeführt werden muss. ❍

Komplexe Aktivitäten unterscheiden sich von komplexen Prozessmustern darin, dass sie keine Subaktivitäten referenzieren, da sie lediglich den Kontext eines Vorgehensschrittes dokumentieren.

Definition 5.4.4 (Komplexe Aktivität)
Eine *komplexe Aktivität* $a = (id, mm_i, mm_r)$ ist ein Tupel aus

- einem eindeutigen Identifikator $id \in \mathbb{ID}$,
- Initialmetamodell $mm_i \in \mathbb{MM}$ und
- Ergebnismetamodell $mm_r \in \mathbb{MM}$,

so dass gilt:

$$\forall m \in \mathbb{M}_{mm_i} \cup \mathbb{M}_{mm_r} : \exists m' \in \mathbb{M}_{mm} : m \trianglelefteq m' \quad ❍$$

Im Folgenden wird der Begriff Prozessmodell auf Basis des Prozessmusteransatzes formal definiert. Ein Prozessmodell besteht demnach aus einem Produktmodell, welches die Struktur der möglichen Modelle der Entwicklungsartefakte definiert, und einer Menge von Vorgehensbausteinen, wie sie bereits eingeführt wurden. Zusätzlich wird festgelegt, dass die Ein- und Ausgabekontexte der Vorgehensbausteine Transformationen auf den Instanzen des Produktmodells definieren.

Definition 5.4.5 (Prozessmodell)
Ein *Prozessmodell* $PM = (mm, EP, EA, CP, CA)$ ist ein Tupel bestehend aus:

- *Produktmodell* $mm \in \mathbb{MM}$
- eine Menge *elementarer Prozessmuster* EP mit

$$\forall p \in EP : P|_R|_{mv_0} = P|_R|_{mv_1} = mm$$

- eine Menge *elementarer Aktivitäten* EA mit

$$\forall a \in EA : A|_R|_{mv_0} = A|_R|_{mv_1} = mm$$

- eine Menge *komplexer Prozessmuster* CP
- eine Menge *komplexer Aktivitäten* CA

Hierbei gilt:

$$\forall m \in \bigcup_{\substack{mm \in CP|_{mm_i} \cup CP|_{mm_r} \cup \\ CA|_{mm_i} \cup CA|_{mm_r}}} \mathbb{M}_{mm} : \exists m' \in \mathbb{M}_{mm} : m \trianglelefteq m' \quad \wedge \tag{5.2}$$

$$\forall R_i, R_j \in (EP \cup EA)|_R \text{ mit } R_i \neq R_j : \\ \forall r_i \in R_i, r_j \in R_j : tx(R_i, r_i) \neq tx(R_j, r_j) \tag{5.3}$$

Weiterhin existiert eine Relation *realizes* die angibt, ob ein Prozessmuster eine Aktivität *realisiert*:

$$realizes : EP \cup CP \times EA \cup CA \rightarrow \mathbb{B}$$

Eine Aktivität wird *ausgeführt*, indem ein sie realisierendes Prozessmuster ausgeführt wird.
Ein Prozessmuster wird *ausgeführt*, indem das Produktmodell entsprechend der Transformationsspezifikation erweitert wird, bzw. indem die in ihm referenzierten Aktivitäten ausgeführt werden. ❍

Durch (5.2) wird festgelegt, dass alle Modelle die durch das Initial- bzw. Ergebnismetamodell komplexer Vorgehensschritte beschrieben werden Teilmodelle der möglichen Instanzen des Produktmodells sind. (5.3) besagt, dass alle Regeln aus Regelwerken unterschiedlicher elementarer Vorgehensschritte unterschiedliche *tx*-Werte aufweisen müssen. Innerhalb eines Vorgehensschrittes haben die Regeln gemäß den Definitionen 5.4.1 und 5.4.2 immer identische *tx*-Werte.

Tabelle 5.3 zeigt ein Beispiel für ein komplexes Prozessmuster zur Erstellung von Artefakten des Business Requirements Modells der KOGITO-Methodik. Die Ausführung dieses Prozessmusters erzeugt ein „Business Reference Model“, welches konform zu dem Metamodell der rechten Regelseite ist. Das Produktmodell mm_{PM}, in dem die Artefakte des Business Reference Modells und deren Beziehungen definiert sind ist in Abbildung 5.2 dargestellt. Durch die Verwendung des Singleton-Symbols für die Klasse „Business Reference Model Worksheet“ wird festgelegt, dass lediglich eine Instanz dieser Klasse im erzeugten Artefaktmodell existieren darf.

In der Beschreibung des Prozessmusters wird innerhalb des UML-Aktivätsdiagramms auf die beiden zu realisierenden Aktivitäten „Create Business Area“ und „Create Process Area“ verwiesen. Die dritte auszuführende Aktion im Aktivitätsdiagramm, „Examine existing Reference Model“, wird innerhalb der Prozessmusterdokumentation textuell beschrieben. Im Rahmen dieser Arbeit soll in erster Linie die formale Spezifikation der Transformationen des Artefaktmodells betrachtet werden. Aus diesem Grund wurde auf die Angabe von textuellen Beschreibungen verzichtet, obwohl diese in der Praxis selbstverständlich ein sehr wesentlicher Bestandteil eines Prozessmusters sind.

Die beiden referenzierten elementaren Aktivitäten sind in den Tabellen 5.4 und 5.5 dargestellt. Die Transformationsspezifikation der Aktivität „Create Business Area“ legt fest, dass in diesem Vorgehensschritt ein bestehendes Dokument vom Typ „Business Reference Model Worksheet“ um ein „Business Area Worksheet“ erweitert wird. Die Aktivität „Create Process Area“ spezifiziert die Erweiterung eines bestehenden „Business Area Worksheet“ um eine neues „Process Area Worksheet“.

Für die Erzeugung neuer Artefakte werden oftmals zusätzliche Informationen, wie typischer Weise ein Name, benötigt. Hierzu bietet sich, wie im Beispiel dargestellt, die Verwendung paramtrisierbarer Rewrite-Regeln für die Spezifikation elementarer Vorgehensschritte an (siehe Definition 3.3.12, S. 77). Somit entsprechen die beiden Aktivitätsbeschreibungen formal gesehen einer (potentiell unendlichen) Menge möglicher Aktivitäten mit jeweils unterschiedlichen konstanten Werten für die Parameter. Referenziert ein komplexes Prozessmuster eine solche Aktivität, so kann aus dieser Menge *eine* dieser Aktivitäten vom Entwickler gewählt werden (siehe Definition 5.4.3).

Die Tabellen 5.6 und 5.7 zeigen Beispiele für elementare Prozessmuster. Durch die Anwendung der Regel BusinessAreaCreation($REF-MODEL, $BUSINESS-AREA) aus Tabelle 5.6 wird für ein bestehendes Dokument des Typs „Business Reference Model Worksheet“ ein neues Formular des Typs „Business Area Worksheet“ erzeugt. Das entsprechende Prozessmuster beschreibt hierzu wie ein neues Geschäftsfeld als Teil des Referenzmodells definiert wird. Die Namen des Referenzmodells und des neuen Geschäftsfeldes werden als Parameter übergeben.

Formal gesehen entspricht die Transformationsregel in Tabelle 5.6 der Menge aller Regeln mit der angegebenen Struktur, die sich jeweils durch konstante Werte an den Stellen der Paramtervariablen

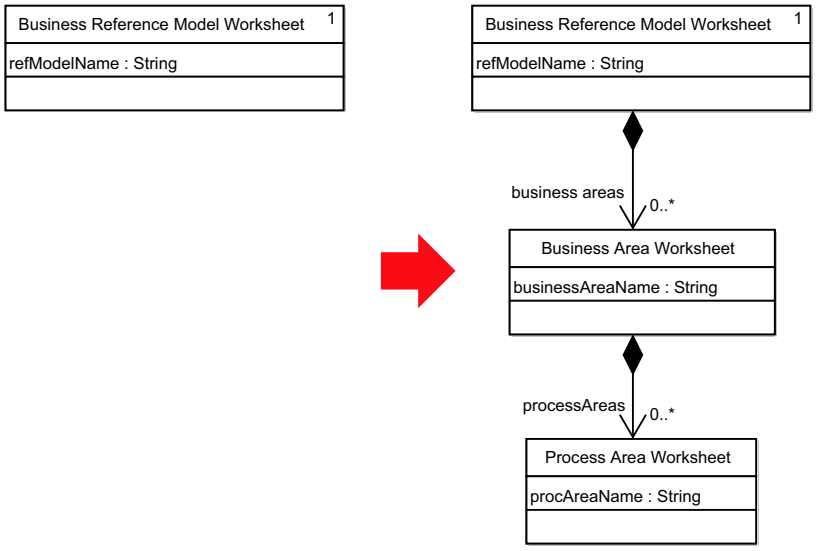
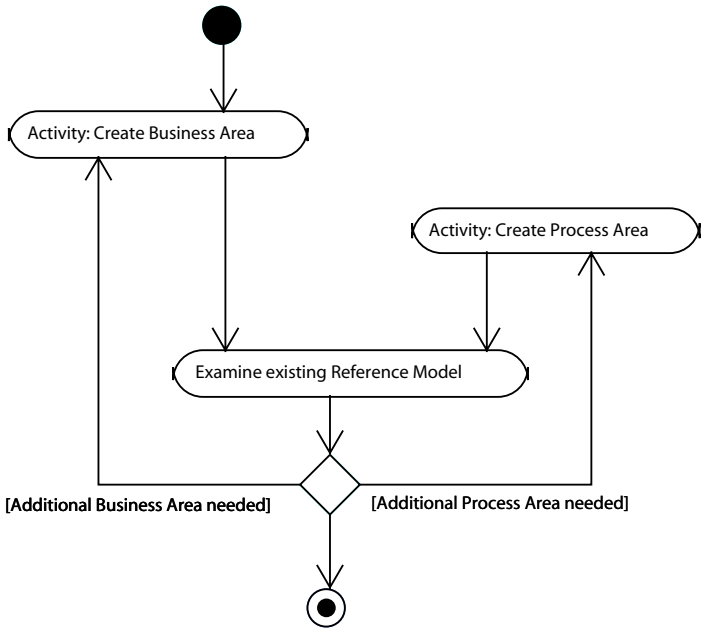
Prozessmuster: Reference Model Creation	
Realisiert:	Create Reference Model
Problem:	...
Projektkontext:	...
Transformation:	 ReferenceModelCreation
Beschreibung:	 Examine existing Reference Model: ...
Anwendungs-beispiele:	...

Tabelle 5.3: Das Prozessmuster „Reference Model Creation“

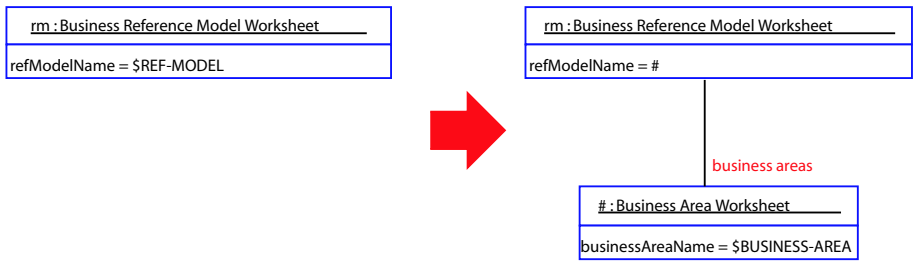
Aktivität: Create Business Area($REF-MODEL, $BUSINESS-AREA)	
	
Transformation:	CreateBusinessArea($REF-MODEL, $BUSINESS-AREA)

Tabelle 5.4: Die Aktivität „Create Business Area“

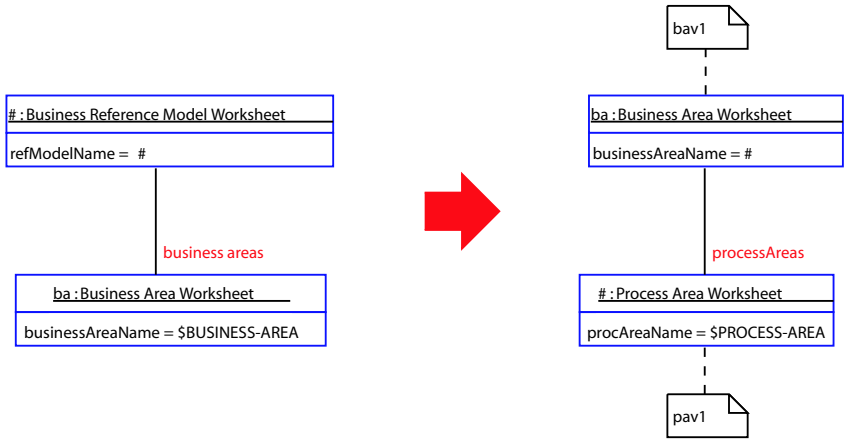
Aktivität: Create Process Area($BUSINESS-AREA, $PROCESS-AREA)	
	
Transformation:	CreateProcessArea($BUSINESS-AREA, $PROCESS-AREA)

Tabelle 5.5: Die Aktivität „Create Process Area“

unterscheiden. Durch Angabe geeigneter Parameter kann zur Ausführungszeit des Musters eine konkrete Regel aus dieser Menge gewählt werden.

Analog hierzu wird durch das Prozessmuster „Process Area Creation“ eine neues Dokument des Typs „Process Area Worksheet“ für ein bestehendes „Business Area Worksheet“ erzeugt.

Wird das Prozessmuster „Reference Model Creation“ ausgeführt, so können zur Ausführung der beiden referenzierten Aktivitäten die Prozessmuster „Business Area Creation“ und „Process Area Creation“ verwendet werden. Gegeben sei beispielsweise ein Modell der Artefakte, welches lediglich aus einem leeren Dokument des Typs „Business Reference Model Worksheet“ mit dem Namen „Electronics Retailer Model“ besteht. Die nachfolgende Liste gibt eine mögliche Folge von Anwendungen der Prozessmuster mit den vorgestellten Transformationsregeln an:

1. BusinessAreaCreation(„Electronics Retailer Model“, „Sales“)
2. BusinessAreaCreation(„Electronics Retailer Model“, „Human Resources“)
3. ProcessAreaCreation(„Sales“, „Payment“)

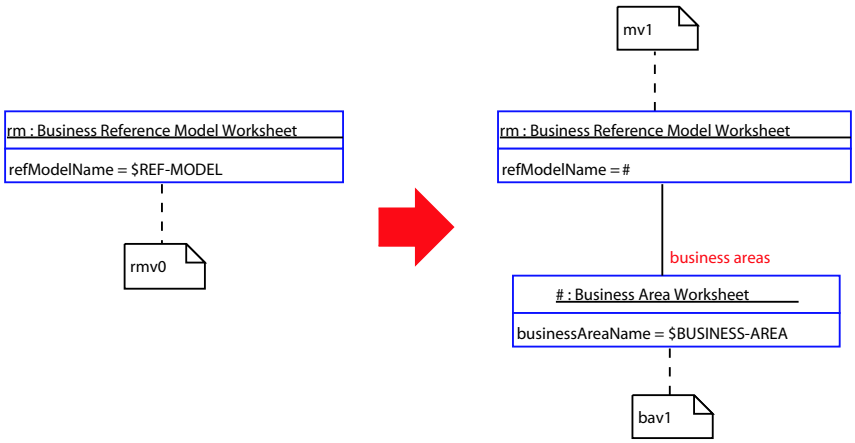
Prozessmuster: Business Area Creation($REF-MODEL, $BUSINESS-AREA)	
Realisiert:	Create Business Area
Problem:	...
Projektkontext:	...
Transformation:	 BusinessAreaCreation($REF-MODEL, $BUSINESS-AREA)
Beschreibung:	...
Anwendungs-beispiele:	...

Tabelle 5.6: Das Prozessmuster „Business Area Creation“

4. ProcessAreaCreation(„Sales“, „Shipment“)
5. ProcessAreaCreation(„Human Resources“, „Recruitment“)

Im Beispiel wird zunächst ein neues Dokument vom Typ „Business Area Worksheet“ angelegt, in dem das Geschäftsfeld „Sales“ beschrieben wird. Anschließend wird ein Dokument gleichen Typs mit für das Geschäftsfeld „Human Resources“ angelegt. In den Schritten 3 und 4 wird jeweils ein „Process Area Worksheet“ für das Geschäftsfeld „Sales“ erzeugt, während der 5. Schritt ein solches Dokument als Teil der Beschreibung des Geschäftsfeldes „Recruitment“ erzeugt. Das vollständige Modell der Artefakte nach der Anwendung der beiden Prozessmuster ist in Abbildung 5.31 dargestellt.

5.4.2 Konsistenz von elementaren Aktivitäten und realisierenden Prozessmustern

Damit ein Prozessmuster eine Aktivität realisieren kann, muss es mit den in der Aktivität angegebenen Eingaben mindestens die dort festgelegten Ausgaben erzeugen (siehe Abschnitt 2.2.2, S. 33 ff.). Werden BOTL-Rewrite-Transformationen zur Spezifikation der Vorgehensschritte verwendet, so bedeutet dies, dass bei Wahl identischer Parameterwerte für eine Aktivität und das sie realisierende Prozessmuster gilt:

1. Ein Modellfragment das von einer Aktivität im Quellmodell gematcht wird, wird von einem Prozessmuster das die Aktivität realisiert entweder ganz oder in Teilen gematcht.
2. Jedes von einem Prozessmuster erzeugte Modellfragment enthält als Teilfragment das Ergebnis der Anwendung der Aktivität, die es realisiert.

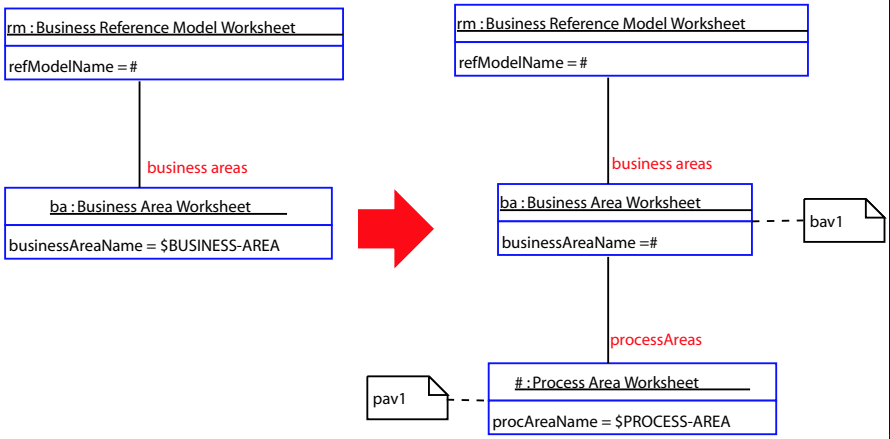
Prozessmuster: Process Area Creation($BUSINESS-AREA, $PROCESS-AREA)	
Realisiert:	Create Process Area
Problem:	...
Projektkontext:	...
Transformation:	 ProcessAreaCreation($BUSINESS-AREA, $PROCESS-AREA)
Beschreibung:	...
Anwendungs-beispiele:	...

Tabelle 5.7: Das Prozessmuster „Process Area Creation“

Definition 5.4.6 (*canRealize*(p,a))
Sei pm ein Prozessmodell, $p \in pm|_{EP}$ ein elementares Prozessmuster und $a \in pm|_{EA}$ eine elementare Aktivität. Das Prozessmuster p kann die Aktivität a realisieren, falls die Relation $canRealize(p,a)$ gilt, mit:

$$
\begin{aligned}
&canRealize(p,a) :\Leftrightarrow \\
&\quad \forall m \in \mathbb{M}_{pm_{mm}} : \bigcup\nolimits_{\mathrm{m}} srcMatches(\{m\}, p|_R)|_{mf} \trianglelefteq \bigcup\nolimits_{\mathrm{m}} srcMatches(\{m\}, a|_R)|_{mf} \wedge \\
&\qquad\qquad rwTransform(\{m\}, a|_R) \trianglelefteq rwTransform(\{m\}, p|_R)
\end{aligned}
$$

❍

Die Definition legt formal den Sachverhalt fest, dass der für beliebige, gültige Artefaktmodelle der Eingabekontext eines Prozessmusters eine Teilmenge des Eingabkontexts der realisierten Aktivität sein muss. Umgekehrt muss der Ausgabekontext des Prozessmusters eine Übermenge dessen der Aktivität sein.

Der nachfolgende Satz gibt eine Reihe von Kriterien an, anhand derer sich nachweisen lässt, dass ein Prozessmuster eine Aktivität realisieren kann.

Satz 5.4.1 (Prozessmuster realisiert Aktivität)
Die Konsistenzbedingungen dafür, dass ein elementares Prozessmuster p eine elementare Aktivität a realisieren kann sind formal erfüllt, falls für jede Regel $r_a \in a|_R$ der Transformationsspezifikation der

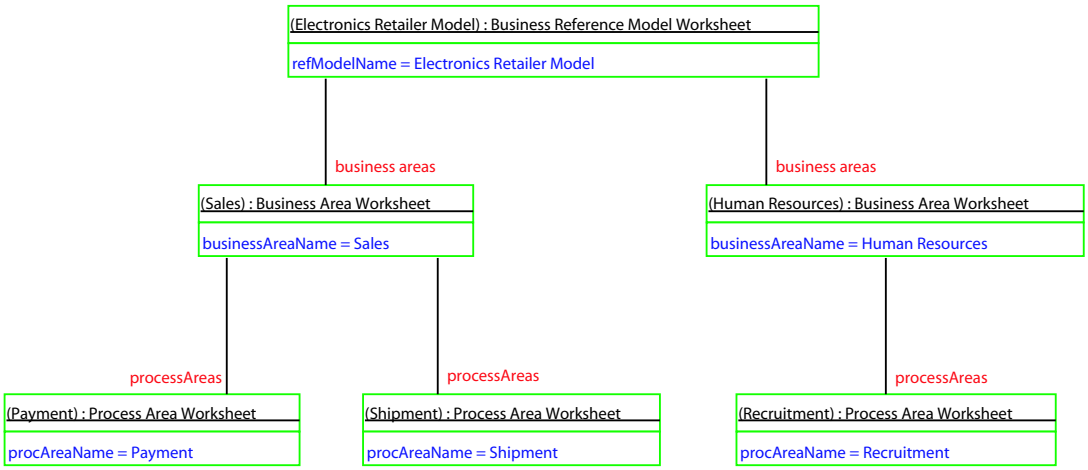


Abbildung 5.31: Das durch die Anwendung von Prozessmustern entstandene Modell der Artefakte

Aktivität eine Regel $r_p \in p|_R$ im Prozessmuster existiert, so dass gilt:

$$r_p|_{mv_0} \sqsubseteq r_a|_{mv_0} \wedge \tag{5.4}$$

$$r_p|_{mv_0} \text{ und } r_a|_{mv_0} \text{ verfügen über identische Constraints über die Objektvariablen von } r_p|_{mv_0}|_{OVB} \text{ und } corresponds(r_p|_{mv_0}|_{OVB}). \tag{5.5}$$

$$OVP(r_a, r_p) = \{OVP\} \text{ mit } OVP = \{(ov_a^0, ov_p^0), \ldots, (ov_a^n, ov_p^n)\} \neq \emptyset \text{ und} \tag{5.6}$$

$$\forall ov_a^i \in r_a|_{mv_1}|_{OVB} : \exists (ov_p^i, ov_a^i) \in OVP : corresponds(ov_a^i) = ov_p^i \tag{5.7}$$

❑

Der Satz geht davon aus, dass sich die Regeln in Prozessmustern und Aktivitäten paarweise zuordnen lassen. (5.4) und (5.5) stellen sicher, dass das Prozessmuster denselben Kontext wie die Aktiviät matcht. (5.6) und (5.7) besagt, dass jedes durch die Aktivität erzeugte Fragment eine Substruktur des durch das Prozessmuster erzeugten Fragments ist. Hierzu kann auf den in Abschnitt 4.2 Entwickelten *OVP*-Mechanismus zurückgegriffen werden (siehe Definition 4.2.10, S. 137). Die hier angegebenen Kriterien sind somit hinreichend, jedoch nicht notwendig, um die gewünschte Konsistenzbedingung nachzuweisen.

Beweis-Skizze zu Satz 5.4.1, Seite 223: (5.4) hält fest, dass die Struktur der Quellmodellvariablen des Prozessmusters eine Substruktur derer der Aktiviät sein muss. Daher ist gemäß Lemma 4.2.2 sichergestellt, dass jedes von r_a gematchte Modellfragment zumindest in Teilen auch von r_p gematcht wird.

Da laut (5.5) alle Quellobjektvariablen von r_p über dieselben Constraints wie die in r_a verfügen ist sichergestellt, dass genau dann wenn wenn r_a ein Fragment ungleich $(\emptyset, \emptyset)$ liefert dies auch für die r_p gilt (siehe Definition 3.5.5, S. 89).

(5.6) stellt sicher, dass $OVP(r_a, r_p)$ eine einelementige Lösung liefert. Dies impliziert bereits, dass gilt: $r_a|_{mv_1} \sqsubseteq r_p|_{mv_1}$ (siehe Definition 4.2.10 137), d.h. die Zielmodellvariable der Aktivitätsregel ist eine Substruktur der Zielmodellvariable der Prozsessmusterregel. Weiterhin wird sichergestellt, dass die in der Menge *OVP* enthaltenen Paare aus Objektvariablen jeweils dieselben Objekte erzeugen.

Durch (5.7) wird schließlich festgelegt, dass r_p nicht nur dieselben Objekte wie r_a erzeugt, sondern eine echte Superstruktur des von r_a erzeugten Modellfragments. ❑

Beispiel: Das elementare Prozessmuster aus Tabelle 5.7 realisiert die Aktivität „Create Process Area“ bei gleicher Belegung der Parameter. Das die hierfür notwendigen Bedingungen erfüllt sind lässt sich anhand von Satz 5.4.1 nachweisen:

Forderung (5.4) ist offensichtlich erfüllt, da die Quellmodellvariablen der durch die parametrisierte Regel repräsentierte Regeln alle strukturell isomorph zu den entsprechenden Quellmodellvariablen der Aktivitätsregel darstellen.

Der Constraint über dem Attribut *businessAreaName* in der Quellmodellvariblen ist ebenfalls für alle Regeln identisch (Forderung (5.5)).

Forderung (5.6) ist erfüllt, da gilt:
$OVP(ProcessAreaCreation(),CreateProcessArea()) =$

$$\{\{(ProcessAreaCreation().bav1, CreateProcessArea().bav1),\\ (ProcessAreaCreation().pav1, CreateProcessArea().pav1)\}\}$$

Weiterhin gilt $CreateProcessArea()|_{mv_1} \sqsubseteq ProcessAreaCreation()|_{mv_1}$ mit

$$corresponds(CreateProcessArea().bav1) = ProcessAreaCreation().bav1 \text{ und}\\ corresponds(CreateProcessArea().pav1) = ProcessAreaCreation().pav1$$

Also ist auch Forderung (5.6) von Satz 5.4.1 erfüllt und die Transformationsspezifikation des Prozessmusters „Process Area Creation“ erfüllt formal aller Vorraussetzungen um die Aktivität „Create Process Area“ zu realisieren.

Tabelle 5.6 stellt ein elementeres Prozessmuster dar, welches die Aktivität „Create Business Area“ realisiert. Da die Transformationsregeln der beiden Vorgehensschritte identisch sind, sind die Forderungen von Satz 5.4.1 trivialer Weise erfüllt. ❍

5.4.3 Konsistenz zwischen komplexen und elementaren Prozessmustern

Für die Spezifikation komplexer Prozessmuster die elementare Vorgehensschritte referenzieren, ist es wichtig sicherstellen zu können, dass der gewünschte Ergebniskontext sich mit Hilfe der elementaren Vorgehensschritte erbringen lässt. Dies ist dann der Fall, wenn das durch die Transformationen der elementaren Vorgehensschritte erzeugte Rewrite-Regelwerk immer ein zum Ergebniskontext des komplexen Prozessmusters metamodellkonformes Modell erzeugt. Die nachfolgende Definition fasst diesen Sachverhalt formal:

Definition 5.4.7 (Korrekt realisierte Prozessmuster)
Sei pm ein Prozessmodell und $p \in pm|_{CP}$ ein komplexes Prozessmuster dessen Ausgabekontext durch das Metamodell $p|_{mm_r} \in \mathbb{MM}$ definiert ist und für das gilt:

$$\{A^0,\ldots,A^n\} := p|_{\mathbb{RA}} \subseteq \mathscr{P}(PM|_{EA})$$

Das Prozessmuster p wird *korrekt realisiert*, falls gilt:

$$\forall a^i \in \bigcup_{A \in p|_{\mathbb{RA}}} A : \exists p' \in pm|_{EP} : realizes(p', a^i) \tag{5.8}$$

$$\forall \{a^0, \ldots a^n\} \text{ mit } a^0 \in A^0, \ldots a^n \in A^n,$$
$$\{p^0, \ldots, p^n\} \text{ mit } realizes(p^0, a^0), \ldots realizes(p^n, a^n) :$$
$$mmConform(\bigcup_{p^i \in \{p^0, \ldots, p^n\}} p^i|_R[p_R|_{mv_1}|_{mm}/p|_{mm_r}]) \tag{5.9}$$

❍

(5.8) legt fest, dass zu jeder von dem komplexen Prozessmuster referenzierten Aktivität ein elementares Prozessmuster existieren muss, welches die jeweilige Aktivität realisiert. (5.9) besagt, dass jede Kombination aus Prozessmustern, die jeweils die geforderten Aktivitäten realisieren, ein zu dem Zielmetamodell von P konformes Modell erzeugen müssen. Der Ausdruck $PP|_R[PP_R|_{mv_1}|_{mm}/PP|_{mm_r}]$ substituiert hierbei alle Zielmetamodelle in allen Regeln durch das Zielmetamodell des komplexen Prozessmusters PP.

Gemäß Satz 4.4.2 (S. 172) kann diese Eigenschaft sichergestellt werden, indem gezeigt wird, dass die Transformationsspezifikationen der elementaren Prozessmuster zusammen mit der Identitätsabbildung metamodellkonform sind.

Es wird nun gezeigt, dass durch die Anwendung beiden parametrisierbaren Regeln aus den Tabllen 5.6 und 5.7 ausschließlich Artefaktmodelle erzeugt werden, die konform zu dem durch das Metamodell mm_{RM} spezfizierten Ausgabekontext des komplexen Prozessmusters „Reference Model Creation“ sind. Gemäß Satz 4.4.2 ist hierzu zu zeigen, dass gilt:

(*i*) $R_1 := R^{id}_{mm_{RM}} \cup$BusinessAreaCreation($REF-MODEL, $BUSINESS-AREA) ist metamodellkonform

(*ii*) $R_2 := R^{id}_{mm_{RM}} \cup$ProcessAreaCreation($BUSINESS-AREA, $PROCESS-AREA) ist metamodellkonform

Nachweis der Metamodellkonformität für R_1

Im Folgenden wird der Nachweis erbracht, dass das Regelwerk R_1 ein metamodellkonformes Regelwerk bildet. Die Regel BusinessAreaCreation($REF-MODEL, $BUSINESS-AREA) enthält zwei Parameter und stellt somit eine verkürzte Schreibweise für eine (potentiell unendliche) Menge von Regeln dar, die sich jeweils durch einen konstanten Wert anstelle der Parameter unterscheiden. Dennoch kann der Beweis ohne Weiteres für die gesamte Klasse der durch die parametrisierte Regel repräsentierten Regeln erbracht werden, indem die Parameter als „symbolische“ Konstanten im Beweis behandelt werden. Da keine der Regeln des Identitätsregelwerks konstante Werte aufweist ist der tatsächliche Wert eines Parameters für den Nachweis unbedeutend.

Die Abbildungen 5.32 und 5.32 zeigen die beiden Regeln id_1 und id_2 des Identitätsregelwerks $R^{id}_{mm_{RM}}$.

Zunächst wird die Anwendbarkeit des Regelwerks R_1 nachgewiesen. Gemäß 4.4.7 reicht es hierfür zu zeigen, dass gilt:

1. BusinessAreaCreation($REF-MODEL, $BUSINESS-AREA) ist anwendbar
2. Sei $R :=$ BusinessAreaCreation() $\cup R^{id}_{mm_{RM}}$. Dann muss gelten:

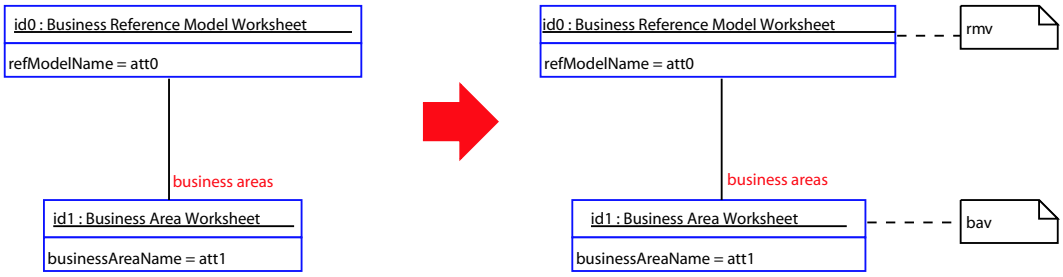


Abbildung 5.32: Die Regel id_1

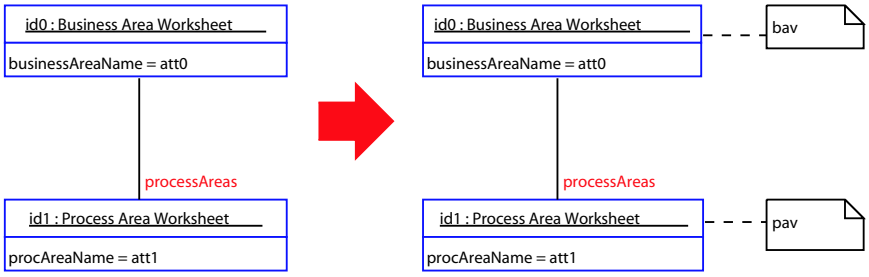


Abbildung 5.33: Die Regel id_2

a) $conflictFree(R, id_1.rmv1, \text{BusinessAreaCreation}().\text{rmv1})$
b) $conflictFree(R, id_1.rmv1, \text{BusinessAreaCreation}().\text{bav1})$
c) $conflictFree(R, id_1.bav1, \text{BusinessAreaCreation}().rmv1)$
d) $conflictFree(R, id_1.bav1, \text{BusinessAreaCreation}().bmv1)$
e) $conflictFree(R, id_2.bav1, \text{BusinessAreaCreation}().rmv1)$
f) $conflictFree(R, id_2.bav1, \text{BusinessAreaCreation}().bav1)$
g) $conflictFree(R, id_2.pav1, \text{BusinessAreaCreation}().rmv1)$
h) $conflictFree(R, id_2.pav1, \text{BusinessAreaCreation}().bav1)$
i) $conflictFree(R, \text{BusinessAreaCreation}().bmv1, \text{BusinessAreaCreation}().bmv1)$

1. Anwendbarkeit von BusinessAreaCreation()

Erzeugen gültiger Modellfragmente

Gemäß Satz 4.1.2 genügt es zu zeigen, dass das Gleichungssystem GL aus Definition 3.5.4 höchstens eine Lösung hat. Dies ist offensichtlich der Fall, die einzige Lösung des Gleichungssystems lautet hierbei:

$$\begin{aligned} mfm^1_\mu|_{match_o}(rmv1)|_{oi} &= mfm^0_\mu|_{match_o}(rmv0)|_{oi} \\ mfm^1_\mu|_{match_o}(rmv1).refModelName &= \$\text{REF-MODEL} \\ mfm^1_\mu|_{match_o}(bav1)|_{oi} &= \diamondsuit \\ mfm^1_\mu|_{match_o}(bav1).businessAreaName &= \$\text{BUSINESS-AREA} \end{aligned}$$

Die paramtrisierbare Regel BusinessAreaCreation() repräsentiert eine Menge von Regeln mit unterschiedlichen konstanten Werten an Stelle der Paramter. Offensichtlich existiert für jede dieser Regeln

eine eindeutige Lösung die sich durch ersetzen der mit einem $-gekennzeichneten Werte durch einen konstanten Wert ergibt.

Deterministische Objektvariablen

Für die Objektvariable $rmv1$ gilt:

$$dependsOn(rmv1, BusinessAreaCreation()) = rmv0$$
$$attDependsOn(rmv1, refModelName, BusinessAreaCreation()) = rmv0$$
$$rmv0 \overset{BusinessAreaCreation()|_{mv_0}}{\rightsquigarrow} rmv0$$

Demnach ist $rmv1$ gemäß Satz 4.1.4 (4.15) deterministisch.

Die Objektvariable $bav1$ hat den Wert $\diamondsuit$ als Identifikator und ist somit gemäß Satz 4.1.4 (4.13) deterministisch.

Konfliktfreie Objektvariable

Die beiden Zielobjektvariablen sind nicht vom gleichen Typ, folglich gilt gemäß Lemma 4.1.5:

$$\neg potConflicting(R, \text{BusinessAreaCreation}().rmv1, \text{BusinessAreaCreation}().bav1)$$
$$\overset{\text{Lem. 4.1.6 (4.19)}}{\Rightarrow} conflictFree(R, \text{BusinessAreaCreation}().rmv1, \text{BusinessAreaCreation}().bav1)$$

Dementsprechend sind sämtliche durch die paramterisierbare Regel BusinessAreaCreation($REF-MODEL, $BUSINESS-AREA) repräsentierten Regeln anwendbar.

2. Konfliktfreiheit der Objektvariablen

Offensichtlich sind Objektvariablen unterschiedlichen Typs immer konfliktfrei (Lemma 4.1.6 (4.19)). Dementsprechend müssen nur die Paare von Objektvariablen mit demselben Typ untersucht werden:

(i) $conflictFree(R, id_1.rmv1, \text{BusinessAreaCreation}().rmv1)$

(ii) $conflictFree(R, id_1.bav1, \text{BusinessAreaCreation}().bav1)$

(iii) $conflictFree(R, id_2.bav1, \text{BusinessAreaCreation}().bav1)$

Da das einzige Attribut von BusinessAreaCreation().$rmv1$ den Wert $\diamondsuit$ hat gilt (i) gilt gemäß Lemma 4.1.6 (4.20).

Weiterhin gilt

$$id_2.bav1|_{oiv} = id0 \not\sim \diamondsuit = \text{BusinessAreaCreation}().bav1|_{oiv}$$
$$\overset{\text{Lemma 4.1.5}}{\Rightarrow} \neg potConflicting(R, id_2.bav1, \text{BusinessAreaCreation}().bav1)$$
$$\overset{\text{Lemma 4.1.4}}{\Rightarrow} conflictFree(R, id_2.bav1, \text{BusinessAreaCreation}().bav1)$$

Demnach gilt auch Aussage (ii). Der Nachweis für Aussage (iii) verläuft hierzu vollkommen analog.

Da jedoch BusinessAreaCreation() eine Menge von Regeln repräsentiert, müssen auch die Objektvariablen innerhalb dieser Menge auf Konfliktfreiheit hin untersucht werden. Der Nachweis der Konfliktfreiheit für diese parametrisierten Objektvariablen ist jedoch ebenfalls durch Lemma 4.1.6 gegeben, da die Objektvariablen den Wert $\diamondsuit$ entweder in der einzigen Attributvariablen ($rmv1$) oder im Identifikatorterm ($bav1$) stehen haben.

Somit sind sämtliche Objektvariablen, unabhängig von den Werten, die für die Parameter in BusinessAreaCreation() gewählt werden konfliktfrei und das Regelwerk insgesamt anwendbar.

Metamodellkonformität

Im Folgenden wird nachgewiesen, dass das Regelwerk

$$R^{id}_{mm_{RM}} \cup \text{BusinessAreaCreation(\$REF-MODEL,\$BUSINESS-AREA)}$$

metamodellkonform ist.

Untergrenzenkonformität

Gemäß Lemma 4.4.6 ist das Regelwerk $R^{id}_{mm_{RM}}$ untergrenzenkonform. Offensichtlich gilt weiterhin:

$$\begin{aligned} & varLbConform(\text{BusinessAreaCreation}()|_{mv_1}) \\ \overset{\text{Satz 4.2.4}}{\Rightarrow}\ & lbConform(\text{BusinessAreaCreation}()) \\ \overset{\text{Lemma 4.2.14}}{\Rightarrow}\ & lbConform(R^{id}_{mm_{RM}} \cup \text{BusinessAreaCreation}()|_{mv_1}) \end{aligned}$$

Obergrenzenkonformität

Die Zielmodellvariable der Regel verfügt Interessant ist hierbei lediglich das Assoziationsende an *rmv*1, da es als einziges eine endliche Obergrenze hat. Das Assoziationsende wird im Folgenden mit ae_0 bezeichnet, die Objektvariablenassoziation dieses Typs in id_1 mit ova_{id1} und die in den Regeln BusinessAreaCreation() ova_{cba}.

Die Menge aller durch die paramtrisierbare Regel BusinessAreaCreation() repräsentierten Regeln wird dargestellt als $\{BusinessAreaCreation_0, BusinessAreaCreation_1, \ldots\}$.

Es gilt:

$$\begin{aligned} \mathbb{OV}^{conf*}_R = \big\{ & \{id_1.rmv1, \text{BusinessAreaCreation}().rmv1\}, \\ & \{id_1.bav1, id_2.bav1\}, \\ & \{BusinessAreaCreation_0.bav1\}, \\ & \{BusinessAreaCreation_1.bav1\}, \ldots \big\} \end{aligned}$$

Die Multiplizitätsobergrenze *ub* hat für das Assoziationsende ae_0 den Wert 1. Anhand von Satz 4.2.3 lässt sich die Obergrenzenkonformität des Regelwerks nachweisen:

$$\begin{aligned} & \max_{OV_k \in \left\{ \begin{array}{l} \{(id_1,bav1),(id_2,bav1)\}, \\ \{(BusinessAreaCreation_0,bav1)\}, \\ \{(BusinessAreaCreation_1,bav1)\},\ldots \end{array} \right\}} \left\{ \sum_{\substack{r_l, ova_m: \\ \exists (r_l,ov_h)\in OV_k: \\ (oppositeEnd(AE_i,ae_j),ov_m)\in ova|_{OVAE} \\ \wedge ova_m|_{OVAT}=AE_i \\ \wedge relevant(R,r_l,ova_m)}} ubVarCard(ova_m, ae_j, r_l) \right\} \\ = \quad & \max\{ubVarCard(ova_{id1}, ae_0, id_1), \\ & \quad ubVarCard(ova_{cba}, ae_0, BusinessAreaCreation_0), \\ & \quad ubVarCard(ova_{cba}, ae_0, BusinessAreaCreation_1), \ldots\} \\ \overset{\substack{\text{Def. 4.2.21} \\ \text{Fälle } (ix) \text{ u. } (xii)}}{=} \quad & \max\{1\} \\ = \quad & 1 \\ \leq \quad & ub = 1 \end{aligned}$$

Der Nachweis für das gegenüberliegende Assoziationsende ist trivial, da die Multiplizitätsobergrenze hier den Wert ∞ hat.

Dementsprechend ist das Regelwerk R_1 metamodellkonform. Der Nachweis der Metamodellkonformität für das Regelwerk

$$R_2 = R^{id}_{mm_{RM}} \cup \text{ProcessAreaCreation(\$BUSINESS-AREA, \$PROCESS-AREA)}$$

verläuft vollkommen analog.

Aufgrund der Metamodellkonformität der beiden Teilregelwerke lässt sich nun gemäß Satz 4.4.2 festhalten, dass das Rewrite-Regelwerk

$$RW := \{\text{BusinessAreaCreation}(\$\text{REF-MODEL}, \$\text{BUSINESS-AREA}), \\ \text{ProcessAreaCreation}(\$\text{BUSINESS-AREA}, \$\text{PROCESS-AREA})\}$$

mit

$$tx(RW, \mathit{BusinessAreaCreation}()) \neq tx(RW, \mathit{ProcessAreaCreation}())$$

ausschließlich metamodellkonforme Modelle erzeugt, d.h. es ist metamodellkonform gemäß Definition 4.2.1.

Somit wurde der Nachweis erbracht, dass die beiden elementaren Prozessmuster „Business Area Creation“ und „Process Area Creation“ den gewünschten Ausgabekontext des Prozessmusters „Reference Model Creation“ erzeugen. Die Überprüfung der Transformationsspezifikationen der referenzierten Aktivitäten anstelle der sie realisierenden Prozessmuster würde im allgemeinen Fall nicht ausreichen um diese Eigenschaft nachzuweisen. Der Grund hierfür ist, dass Prozessmuster zusätzliche Artefakte erzeugen können, die von der Aktivität nicht vorgesehen sind. Durch diese könnte ggf. die Metamodellkonformität des Zielmodells verletzt werden.

5.4.4 Zusammenfassung

Wie hier anhand vergleichsweise einfacher und überschaubarer Beispiele gezeigt wurde, erlauben es die hier vorgestellten Verfahren, Vorgehensmodelle präzise zu spezifizieren und die Konsistenz und Umsetzbarkeit des gewählten Vorgehens nachzuweisen.

So ist es nun möglich die Konsistenzbedingungen zwischen Aktivitäten und Prozessmuster, sowie die zwischen komplexen und elementaren Prozessmustern formal zu verifizieren. In ähnlicher Weise lässt sich sicherstellen, dass durch die Prozessmuster eines Prozessmodells zu keinem Zeitpunkt ein inkonsistentes Modell der Artefakte entstehen kann. Hierzu wird die Menge der Transformationsspezifikationen aller Prozessmuster eines Prozessmodells bezüglich ihrer Metamodellkonformität zum gewählten Produktmodell überprüft.

In der Praxis ist jedoch eine geeignete Werkzeugunterstützung für die automatisierte Verifikation der geforderten Eigenschaften unerlässlich. So ließen sich die bestehenden Defizite bei der Werkzeugunterstützung für den Prozessmusteransatz durch die Integration des in Kapitel 6 vorgestellten BOTL-Werkzeuges zusammen mit dessen Verifikationskomponente beseitigen.

6 Werkzeugunterstützung für BOTL-Transformationen

Um die Umsetzbarkeit und Effektivität der erarbeiteten Konzepte belegen zu können, ist eine Werkzeugunterstützung unerlässlich. So wurde das Fehlen einer lauffähigen Implementierung und der damit verbundene Mangel an praktischen Einsatzszenarien bisher als wesentlicher Nachteil von BOTL identifiziert [Spr03]. Um diesem Mangel zu begegnen wurde im Rahmen der vorliegenden Arbeit ein Werkzeug zur graphischen Spezifikation, Verifikation und Ausführung von BOTL-Regelwerken realisiert. Im Verlauf dieses Kapitels wird ein Überblick über das Werkzeug gegeben und eine Reihe verwandter Ansätze vorgestellt.

6.1 Anforderungen und Überblick über das Werkzeug

Das hier vorgestellte BOTL-Werkzeug erlaubt es, BOTL-Regelwerke graphisch zu spezifizieren, die Anwendbarkeit und Metamodellkonformität dieser Regelwerke automatisiert zu verifizieren und Transformationen von Modellen anhand dieser Spezifikationen durchzuführen. Ein solches Werkzeug bildet somit die notwendige Grundlage für eine umfassende Werkzeugunterstützung eines modellbasierten Entwicklungsprozesses, wie er innerhalb dieser Arbeit vorgestellt wird. Im Folgenden werden die wesentlichen Anforderungen an das erstellte Werkzeug skizziert und ein Überblick über seine grundsätzliche Funktionsweise und Architektur gewährt.

6.1.1 Anforderungen an die Werkzeugunterstützung

Es existieren zwei wesentliche nicht-funktionale Anforderungen an das das BOTL-Werkzeug: Zum einen soll die Anwendung plattformunabhängig einsetzbar sein, was Java als Implementierungsplattform nahe legt. Zum anderen sollte das Werkzeug frei verfügbar sein. Hierdurch verbietet sich der Einsatz von kommerziellen COTS-Produkten. Die wichtigsten funktionalen Anforderungen an die BOTL-Werkzeugunterstützung werden nachstehend knapp zusammengefasst:

Spezifikation von BOTL-Transformationen

Das Werkzeug muss es ermöglichen Quellmetamodelle, das Zielmetamodell und Transformationsregeln graphisch zu spezifizieren.

- Ziel- und Quellmetamodelle müssen graphisch spezifizierbar sein. Für die Spezifikation von Metamodellen sind UML-Klassendiagramme zu verwenden. Diese müssen um die Möglichkeit erweitert werden Primärschlüsselattribute für Klassen zu definieren.

- Regeln müssen gemäß dem in Abschnitt 3.4 (S. 78ff) eingeführten UML-Profil für BOTL-Regelwerke graphisch spezifizierbar sein. Die graphisch angepasste „Pfeil-Notation“ ist aufgrund ihrer besseren Lesbarkeit einer Lösung mit Stereotypen vorzuziehen.

- Die syntaktische Korrektheit von Regeln soll bereits während der Eingabe sichergestellt werden. Hierzu muss durch das Modellierungswerkzeug dafür gesorgt werden werden, dass Regeln nur aus zu ihrem Metamodell konsistenten Modellvariablen aufgebaut sind.
- Regelwerke müssen zusammen mit Layout-Informationen und Kommentaren persistent speicherbar sein.
- Regelwerke müssen in einem XML-Format exportiert werden können.

Verifikation von Eigenschaften von BOTL-Regelwerken

Die manuelle Verifikation der Metamodellkonformität eines Regelwerks ist zumeist mit einem sehr hohen Aufwand verbunden. Daher muss es möglich sein, diese Verifikation automatisiert durchzuführen und in einer für den menschlichen Benutzer nachvollziehbaren Weise zu dokumentieren. Die Anforderungen hierzu umfassen im Einzelnen:

- Es muss möglich sein, die Anwendbarkeit von Regelwerken automatisiert zu verifizieren.
- Es muss möglich sein, die Metamodellkonformität von Regelwerken automatisiert zu verifizieren.
- Der Nachweis von Eigenschaften von Regelwerken muss in Form einer für den Benutzer nachvollziehbaren Ausgabe des Werkzeuges vorliegen.
- Fehler in Regelwerken sollen gesammelt werden und dem Benutzer sowohl in Form eines Protokolls als auch graphisch im Editor zur Spezifikation der Regelwerke dargestellt werden.

Transformation von Modellen

Das Werkzeug muss es ermöglichen, einmal spezifizierte Transformationen mit verschiedenen Arten von objektorientierten Modellen durchzuführen. Um den Effekt von Regelwerksanwendungen besser testen zu können, soll das Werkzeug auch einen graphischen Demonstrationsmodus bereitstellen.

- Die Spezifikationen von BOTL-Transformationen müssen verwendet werden können, um zur Laufzeit Transformationen zwischen Modellen durchführen zu können.
- Es sollen unterschiedliche technische Formate von Quell- und Zielmodellen, wie z.B. Java-Objektstrukturen oder XMI-Dokumente, unterstützt werden.
- Die für die Transformation zuständige Komponente soll in der Lage sein, mathematische Ausdrücke in Attributen und Identifikatoren von Objektvariablen zu interpretieren und zur Laufzeit auszuwerten.
- Constraints in Quellmodellvariablen, die sich durch Terme in Objektvariablenidentifikatoren und -attributen ergeben, müssen zur Laufzeit berücksichtigt werden.
- Ein Demonstationsmodus des Modellierungswerkzeugs soll es erlauben, Quellmodelle graphisch zu spezifizieren und und anhand eines BOTL-Regelwerks in ein Zielmodell zu transformieren. Das Ergebnis der Transformation soll ebenfalls graphisch in Form eines Objektmodells dargestellt werden.

6.1.2 Prinzipielle Funktionsweise und Grobarchitektur

An dieser Stelle wird der Lösungsansatz für die Realisierung einer Werkzeugunterstützung für BOTL grob skizziert. Für die Modellierung von Metamodellen und Regelwerken wurde das UML-Werkzeug ArgoUML [Col04a, RVR+03] erweitert. Die so erstellten Spezifikationen können zur Laufzeit durch einen Interpreter ausgeführt werden.

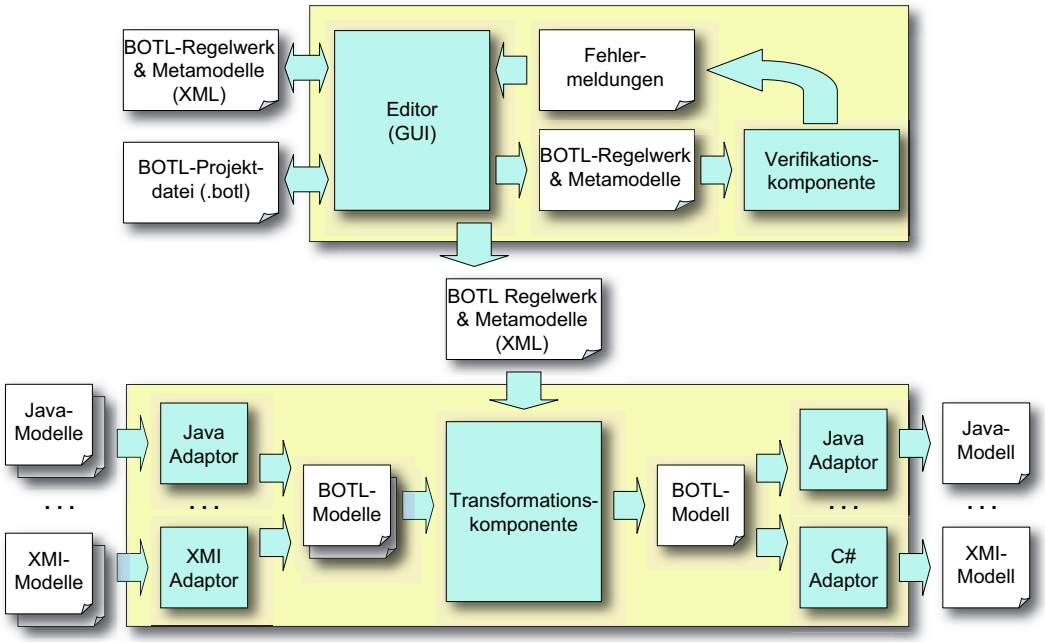


Abbildung 6.1: Überblick über das BOTL-Werkzeug

Abbildung 6.1 veranschaulicht die prinzipielle Funktionsweise und den Aufbau der Werkzeugunterstützung. Diese gliedert sich in zwei Teile: Der obere Teil der Abbildung zeigt die Werkzeugunterstützung zur Spezifikation von Transformationen, während in der unteren Bildhälfte der Interpreter zur Ausführung der Transformationen skizziert ist.

Der Benutzer hat die Möglichkeit mit dem erweiterten ArgoUML-Werkzeug graphisch Metamodelle und Regeln zu spezifizieren. Die Metamodellkonformität eines so erstellten Regelwerks wird durch eine Verifikationskomponente überprüft und evtl. auftretende Fehler an den Benutzer gemeldet. Regelwerke können entweder mit allen graphischen Informationen in Form einer Projektdatei oder in einem XML-Format gespeichert werden.

Die durch den Editor erzeugten XML-Dokumente werden von einer Transformationskomponente eingelesen und interpretiert, um die gewünschten Transformationen für konkrete Modelle auszuführen. Mit Hilfe geeigneter Adapter werden Modelle in unterschiedlichen technischen Formaten für den Transformator verfügbar gemacht und die Ergebnisse einer Modelltransformation wieder in ein entsprechendes Ausgabeformat umgewandelt.

Die für die Spezifikation und Ausführung von Modelltransformationen relevanten Aspekte sind innerhalb des Werkzeuges in einer Reihe von Java-Paketen gekapselt. Abbildung 6.2 gibt einen Überblick über die Paketstruktur der Werkzeuges und die gegenseitige Abhängigkeiten zwischen den Paketen. Die wesentlichen Java-Pakete, aus denen sich die Anwendung zusammensetzt, sind im Einzelnen:

`de.tum.in.botl.metamodel` In diesem Paket finden sich alle Klassen und Interfaces für die

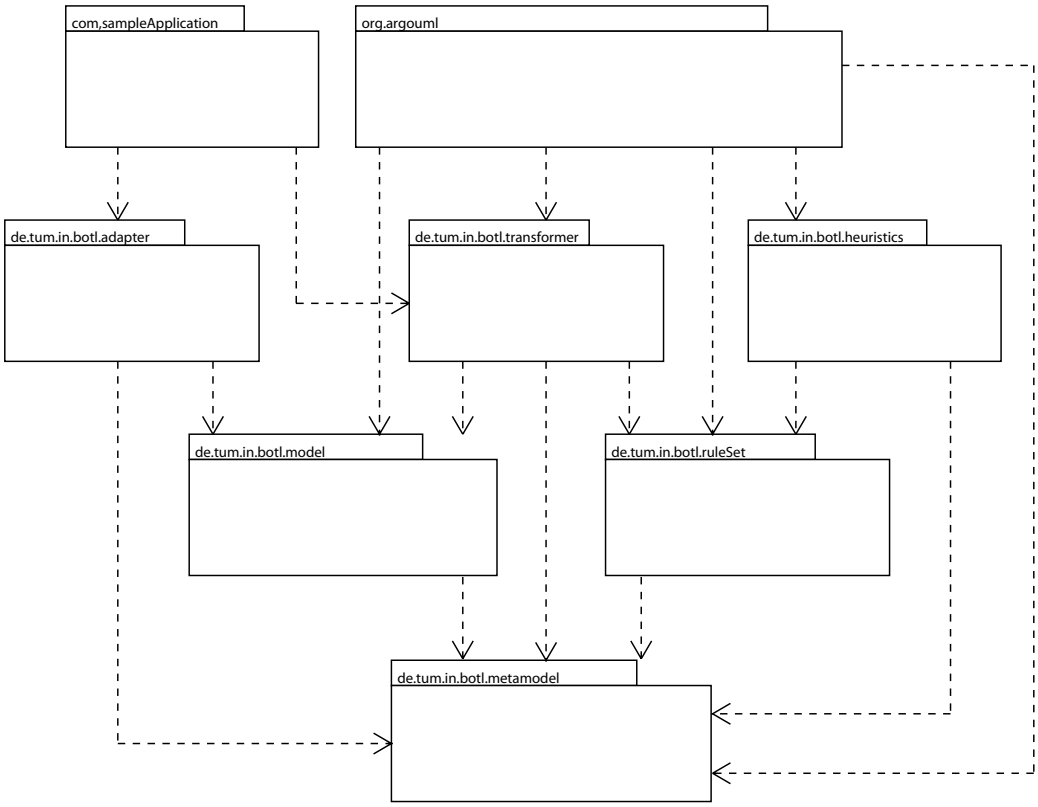


Abbildung 6.2: Die Abhängigkeiten zwischen den Komponenten des Prototypen

werkzeuginterne Darstellung von Metamodellen.

`de.tum.in.botl.model` Das `model`-Paket enthält alle Klassen zur Repräsentation von BOTL-Modellen. Jedes Modell referenziert genau ein Metamodell, das seinen Typ festlegt. Elemente von BOTL-Modellen verweisen ebenfalls auf ihre Typen in diesem Metamodell.

`de.tum.in.botl.ruleSet` Innerhalb dieses Pakets finden sich alle Klassen zur Darstellung von Regelwerken. Da Quell- und Zielmetamodelle für ein Regelwerk bekannt sein müssen, verwendet das Paket auch das `metamodel`-Paket.

`de.tum.in.botl.heuristics` Das Paket kapselt die Klassen für die Verifikation von Eigenschaften von BOTL-Regelwerken. Hierfür stützt sich das Paket auf die Pakete zur Darstellung von Metamodellen und Regelwerken.

`de.tum.in.botl.transformer` Das `transformer`-Paket enthält die für die Transformation von BOTL-Modellen notwendigen Klassen. Demzufolge verwendet es die Pakte zur Repräsentation von Modellen und Regelwerken. Um Instanzen eines Zielmetamodells erzeugen zu können, wird zusätzlich das Paket für die Darstellung von Metamodellen verwendet.

`org.argouml` Dieses Paket und seine Unterpakete enthalten den Code des erweiterten ArgoUML-Werkzeugs. Die Erweiterungen erlauben die graphische Spezifikation von Metamodellen und Regeln. Von der graphischen Oberfläche aus lässt sich die Verifikation der Metamodellkonformität so modellierter Regelwerke anstoßen. Ein Demonstrationsmodus erlaubt es zusätz-

lich, Quellmodelle graphisch zu erstellen und zu transformieren. Hierzu verwendet das Paket `argouml` die Pakete für die Repräsentation von Metamodellen, Regelwerken und Modellen. Für die Transformation von Modellen und Verifikation von Regelwerken werden zusätzlich die Pakte `transformer` und `heuristics` benötigt.

`de.tum.in.botl.adpater` Das Paket enthält Importer und Exporter zum Ein- und Auslesen von Objektgeflechten in unterschiedlichen technischen Formaten. Derzeit existiert je ein Importer und Exporter für Java-Objektgeflechte. Für die Transformation von BOTL-Modellen in technische Formate und umgekehrt werden die Pakete zur Repräsentation von Modellen und Metamodellen benötigt.

Das Paket `com.sampleApplication` in der Abbildung stellt exemplarisch eine Anwendung dar, die das BOTL-Werkzeug für die Transformation von Modellen verwendet. Hierfür werden lediglich die beiden Pakte `adapter`, für die Anbindung an ein technisches Ein-/Ausgabeformat, und `transformer`, für die Ausführung der Transformation benötigt. Das Regelwerk zu Spezifikation dieser Transformation erhält die Transformationskomponente zur Laufzeit in Form eines XML-Dokuments.

6.2 Datenstrukturen

Für die Repräsentation von Metamodellen, Regelwerken und Modellen bzw. Modellfragmenten innerhalb der Werkzeugunterstützung müssen diese zuvor formal definierten Modelle auf ein objektorientiertes Datenmodell abgebildet werden. Abbildung 6.3 zeigt die wichtigsten Klassen und Beziehungen des Java-Modells, welches die BOTL-Konzepte innerhalb des Werkzeugs wiederspiegelt, in Form eines UML-Klassendiagramms.

Das Klassendiagramm beschränkt sich auf die fachlich relevanten Attribute und Assoziationen der Klassenstruktur. Weitere Details, wie beispielsweise Hash-Tabellen, die der Leistungsoptimierung dienen, werden an dieser Stelle nicht berücksichtigt. Auch die Methoden der Klassen sind zur besseren Verständlichkeit des Diagramms nicht abgebildet. Dennoch erfolgt der Zugriff auf sämtliche Attribute und die Navigation über Assoziationen über geeignete Getter- und Setter-Methoden. Die einzelnen Elemente der drei Pakete, welche die Infrastruktur des Werkzeuges bilden, werden in den nachfolgenden Abschnitten eingehender diskutiert.

6.2.1 BOTL-Metamodelle

Das Paket `de.tum.in.botl.metamodel` enthält alle Klassen die nötig sind, um BOTL-Metamodelle zu bilden. Sämtliche Elemente eines Metamodells verfügen über ein Attribut `id` des Typs `String`, welches unter anderem für die Speicherung des Metamodells im XML-Format benötigt wird.

`Metamodel` Ein Objekt dieser Klasse repräsentiert ein Metamodell im Sinne von Definition 3.1.9 (S. 56). Über das Attribut `name` der Klasse wird ein Metamodell referenziert. Weiterhin umfasst eine Instanz der Klasse `Metamodell` je eine Menge von Objekten des Typs `Type`, `BOTLClass` und `ClassAssociation`.

`Type` Instanzen dieser Klasse repräsentieren einen primitiven Typ gemäß Definition 3.1.2 (S. 51). Das Attribut `it` enthält den Namen des jeweiligen Typs in Form eines Strings.

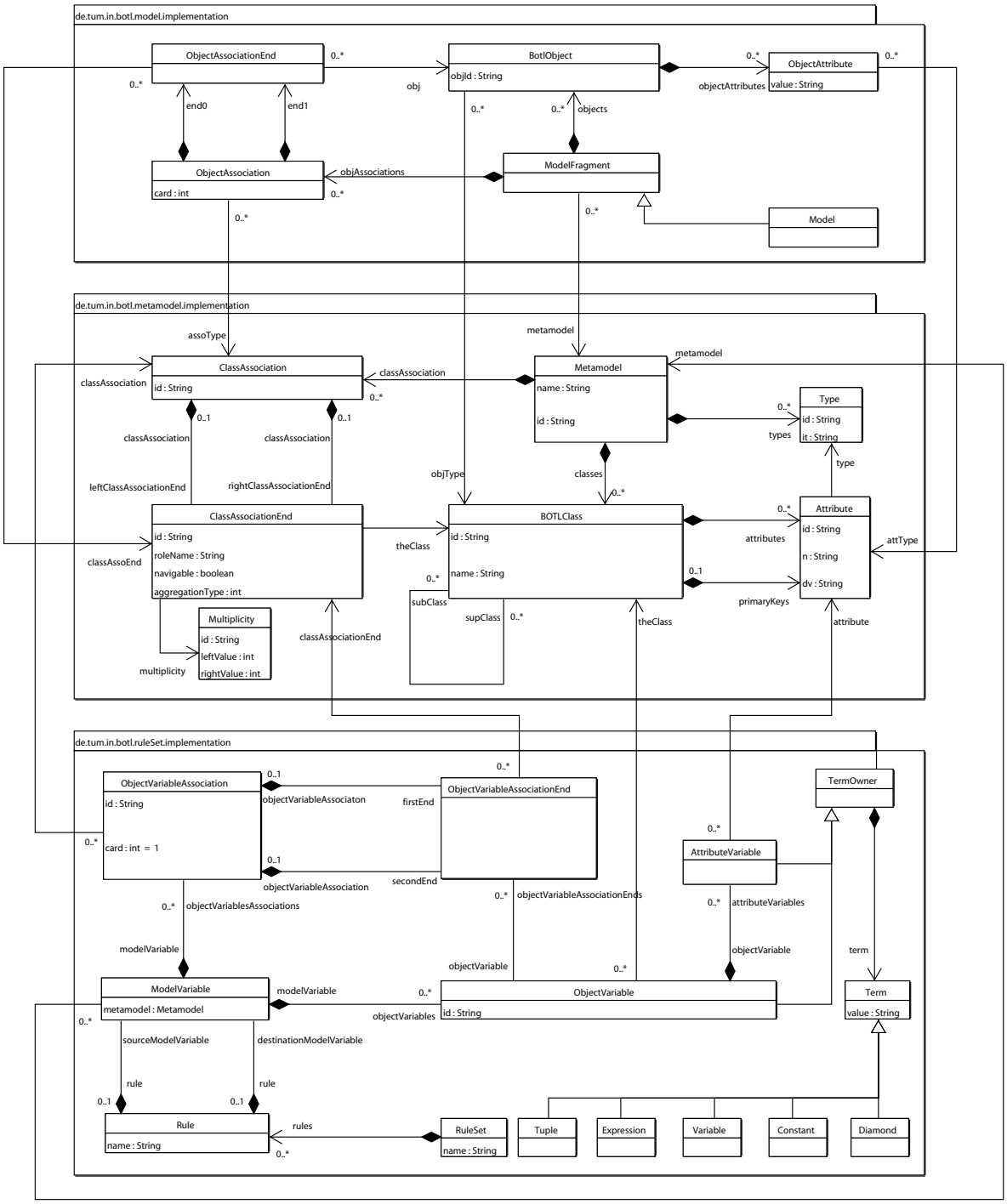


Abbildung 6.3: Die Pakete und Klassen für Metamodelle, Modelle und Regelwerke

`BOTLClass` Objekte der Klasse `BOTLClass` stellen Klassen wie sie in Definition 3.1.4 (S. 52) eingeführt wurden dar. Der Name der Klasse findet sich im Attribut `name`. Darüber hinaus referenziert eine Klasse eine Menge von Super- und Subklassen. Über die Assoziation `attributes` werden alle Attribute der Klasse referenziert. Durch den Aufruf der Methode `getAllAttributes()` erhält man zusätzlich zu den eigenen Attributen einer Klasse auch alle geerbten Attribute der Klasse.

`Attribute` Attribute einer BOTL-Klasse werden auf Instanzen der Klasse `Attribute` abgebildet. Das Attribut `n` enthält den Namen des Attributs, `dv` seinen Standardwert (engl. Default Value) und die Assoziation `type` einen eindeutigen Verweis auf den Typ des Attributs. Jeder von einem Attribut referenzierte Typ muss auch in dessen Metamodell enthalten sein.

`ClassAssociation` Jedes Objekt dieses Typs stellt eine Klassenassoziation im Sinne von Definition 3.1.7 (S. 54) dar. Im Gegensatz zu Defintion 3.1.7 verfügt eine Klassenassoziation nicht über eine Menge von Assoziationsenden der Mächtigkeit 1 oder 2, sondern verweist über die Assoziationen `leftAssociationEnd` und `rightAssociationEnd` jeweils auf ein Objekt des Typs `ClassAssociationEnd`.

`ClassAssociationEnd` Objekte dieser Klasse entsprechen Klassenassoziationsenden von Klassenassoziationen. Die Klasse `ClassAssociationEnd` verfügt über die folgenden Attribute und ausgehenden Assoziationen:

- **`aggregationType`:** Legt die Art der Aggregation („keine“, „Aggregation“, „Komposition“) fest
- **`navigable`:** Ein Boolscher Wert, der angibt, ob die Instanzen der Assoziation in Richtung dieses Endes navigierbar sind.
- **`roleName`:** Der Rollenname des Assoziationsendes
- **`BOTLClass`:** Referenz auf die Klasse an deren Instanzen (oder Instanzen von Unterklassen) Assoziationsenden dieses Typs auftreten dürfen.
- **`multiplicity`:** Verweis auf ein Objekt der Klasse `Multiplicity`, in welchem die Multipliziätsober- und -untergrenzen des Endes festlegt sind.

Multiplicity Diese Klasse enthält jeweils die Multipliziätsober- und -untergrenzen eines Assoziationsendes (`leftValue` und `rightValue`).

6.2.2 BOTL-Regelwerke

Alle Elemente für die interne Repräsentation der in Abschnitt 3.3 (S. 70ff) eingeführten BOTL-Regelwerke finden sich in dem Java-Paket `ruleSet`. Wie die Klassen des Metamodells verfügen auch alle Klassen dieses Pakets über ein Attribut `id` vom Typ `String`, welches als Identifikator für den XML-Export und Import von Regelwerken benötigt wird. Die wesentlichen Klassen dieses Pakets werden im Folgenden knapp vorgestellt:

`RuleSet` Instanzen der Klasse `RuleSet` repräsentieren Regelwerke wie sie in Definition 3.3.10 (S. 75) eingeführt wurden. Ein Regelwerk umfasst eine Menge von Objekten der Klasse `Rule`.

`Rule` Ein `Rule`-Objekt entspricht einer BOTL-Regel (gemäß Def. 3.3.9, S. 75). Sie verweist auf genau eine Quellmodellvariable (`sourceModelVariable`) und eine Zielmodellvariable (`destinationModelVariable`).

`ModelVariable` Objekte dieses Typs stellen Modellvariablen (siehe Def. 3.3.8, S. 74) dar. Jede Modellvariable hat eine `metamodel`-Referenz auf ihr Metamodell, wobei sämtliche Zielmodellvariablen auf dasselbe Zielmetamodell verweisen, während Quellmodellvariablen nie auf das Zielmetamodell verweisen dürfen. Weiterhin enthalten Objekte dieser Klasse jeweils Referenzen auf alle in der Modellvariablen enthaltenen Objektvariablen (siehe Klasse `ObjektVariable`) und alle in ihr enthaltenen Klassenassoziationen (siehe Klasse `ObjectVariableAssociation`).

`ObjectVariable` Die Klasse definiert die werkzeuginterne Struktur von Objektvariablen gemäß Definition 3.3.4 (S. 71). Objekte dieser Klasse besitzen eine Menge von Objekten der Klasse `AttributeVariable` und einen Verweis auf ihren jeweiligen Typ (`theClass`). Weiterhin wird auf alle Objektvariablenassoziationsenden verwiesen, die an der Objektvariable enden (`objectVariableAssociatonEnds`).

`AttributeVariable` Ein `AttributeVariable`-Objekt repräsentiert eine Attributvariable einer Objektvariablen. Der Typ der Variable ist durch einen Verweis auf genau ein Objekt der Klasse `Attribute` des Metamodell-Pakets festgelegt.

`TermOwner` Diese abstrakte Klasse wird von den beiden Klassen `ObjectVariable` und `AttributeVariable` geerbt, um einen einheitlichen Umgang mit Termen in Identifikatoren und in Attributen von Objektvariablen zu ermöglichen. Sie enthält einen Verweis `term` auf den zur jeweiligen Instanz gehörigen Term.

`Term` Objekte der Klasse `Term` bezeichnen Terme. Im Gegensatz zu dem in Abschnitt 3.3 vorgestellten Formalismus wird innerhalb des Werkzeuges nicht explizit zwischen Termen über Identifikatoren und primitiven Typen unterschieden. Der Inhalt eines Terms ist in Form eines `String`-Objekts in dem Attribut `value` festgelegt. Von der Klasse Term erben fünf weitere Klassen, welche jeweils verschiedene Ausprägungen von Termen charakterisieren:

`Diamond` entspricht dem $\diamondsuit$-Wert im Formalismus, der im Werkzeug durch das ♯-Symbol repräsentiert wird.

`Constant` legt fest, dass der `value`-Wert als konstanter Wert zu interpretieren ist.

`Variable` legt fest, dass der `value`-Wert als Variable zu interpretieren ist.

`Expression` entspricht einem mathematischen Ausdruck. Dieser kann über konstante Werte und Variablenwerte verfügen, die in der Regel durch arithmetische Operatoren verknüpft sind. Folglich muss der `value`-Wert einer `Expression` bei Bedarf geparst und ausgewertet werden.

`Tuple` ist ein Tupel, welches selbst wieder aus einer Reihe von Termen bestehen darf. Die Terme, welche selbst auch wieder Tupel sein können, werden im Editor in eckigen Klammern dargestellt (z.B. `[var1*var2, 17, "Test"]`). Tupel dürfen lediglich als Terme von Objektidentifikatoren verwendet werden.

Werkzeugintern wird zwischen drei Typen für Terme unterschieden, für die jeweils unterschiedliche Operationen unterstützt werden:

Arithmetische Ausdrücke werden intern als Java-`float` Werte verarbeitet.

Boolsche Ausdrücke werden intern als Java-`boolean` Werte verarbeitet.

Zeichenketten werden intern als Java-`String` Objekte verarbeitet. Für sie steht lediglich der Konkatenationsoperator „+“ als mögliche Operation zur Verfügung.

`ObjectVariableAssociation` Instanzen dieser Klasse repräsentieren Objektvariablenassoziationen, die in Definition 3.3.6 (S. 73) eingeführt wurden. Im Gegensatz zu der dort vorgestellten Definition verfügen diese im Werkzeug jedoch nicht über eine Menge von Objektvariablenassoziationsenden, sonder über genau zwei Referenzen (`leftClassAssociationEnd` und `rightClassAssociationEnd`) auf jeweils ein Objektvariablenassoziationsende. Der Typ einer Objektvariablenassoziation wird durch einen Verweis auf ein Objekt der Klasse `ClassAssociation` des Metamodell-Pakets festgelegt.

`ObjectVariableAssociationEnd` Die Enden von Objektvariablenassoziationen werden durch Instanzen der Klasse `ObjectVariableAssociationEnd` repräsentiert. Der Typ des jeweiligen Endes wird durch einen Verweis auf ein Objekt der Klasse `ClassAssociationEnd` aus dem Metamodell-Paket festgelegt. Hierbei müssen immer Klassenassoziationsenden referenziert werden, deren Typ zu der zugehörigen Objektvariablenassoziation (wird als `objectVariableAssociation` referenziert) passt. Eine Referenz `objectVariable` legt fest, mit welchem Objektvariablen das jeweilige Ende verbunden ist.

6.2.3 BOTL-Modelle

Die Klassen zur Darstellung von BOTL-Modellen innerhalb des Werkzeuges finden sich im Paket `de.tum.in.botl.model.implementation`. Um Modelle in anderen technischen Formaten, wie z.B. XMI, transformieren zu können, müssen diese zunächst mit Hilfe eines Adapters in das BOTL-interne Format transformiert werden. Die wesentlichen Klassen für die Verarbeitung von BOTL-Modellen durch das Werkzeug werden im Folgenden kurz zusammengefasst:

`ModelFragment` Instanzen dieser Klasse repräsentieren Modellfragmente gemäß Definition 3.5.1 (S. 83). Ein Modellfragment besteht aus einer Menge von BOTL-Objekten (Referenz `objects`) und einer Menge von Objektassoziationen zwischen diesen Objekten (Referenz `objAssociations`). Eine Referenz auf ein Metamodell des Metamodell-Pakets legt fest, zu welchem Metamodell das jeweilige Modellfragment konform ist.

`Model` Die Klasse `Model` erbt alle ihre Eigenschaften von der Klasse `ModelFragment`. Sie wird verwendet, falls ein Modellfragment nicht nur konform zu einem Metamodell ist, sondern eine gültige Instanz des Metamodells darstellt (sieh Def. Def:Modell, S. 61).

`BOTLObject` Diese Klasse wird für die Repräsentation von BOTL-Objekten (siehe Def.) innerhalb des Werkzeugs verwendet. Über die Referenz `objectAttributes` wird auf die Attribute des BOTL-Objekts verwiesen. Der Typ des BOTL-Objekts wird durch eine Referenz auf ein Objekt der Klasse `BOTLClass` festgelegt. Das BOTL-Objekt muss für jedes Attribut der BOTL-Klasse und für jedes von der BOTL-Klasse geerbte Attribut eine Referenz auf ein Objektattribut entsprechenden Typs haben.

`ObjectAttribute` Attribute von BOTL-Objekten werden jeweils durch Instanzen der Klasse `ObjectAttribute` dargestellt. Das Attribut `value` enthält den Wert des Attributes in Form einer Zeichenkette, während sein Typ durch die Referenz `attType` auf ein Objekt des Typs `Attribute` bestimmt wird.

`ObjectAssociation` Objekte dieser Klasse stellen Objektassoziationen gemäß Definition 3.1.13 (S. 59) dar. Auch hier unterscheidet sich die Implementierung geringfügig vom Formalismus, da anstelle einer Menge von Objektassoziationsenden zwei explizite Referenzen (`end0` und `end1`) auf die Enden verwendet werden. Der Typ der Assoziation wird durch eine Referenz `assoType` auf die entsprechende Klassenassoziation festgelegt. Die referenzierten Enden müssen konsistent zum diesem Typ sein. Das Attribut `card` vom Typ `int` gibt die Kardinalität der Assoziation an.

`ObjectAssociationEnd` Objektassoziationsenden referenzieren das BOTL-Objekt (`obj`), mit dem sie verbunden sind und ein Klassenassoziationsende (`classAssoEnd`), um ihren Typ zu spezifizieren.

6.3 Der BOTL-Editor

Für die graphische Spezifikation von Regelwerke und Metamodelle wurde das UML-basierte Modellierungswerkzeug ArgoUML erweitert. Diese Arbeiten entstanden zu großen Teilen im Rahmen eines studentischen Systementwicklungsprojektes; eine detaillierte Beschreibung findet sich in [Kov03].

ArgoUML ist ein unter BSD-Lizenz entwickeltes UML-CASE-Werkzeug. Im Gegensatz zu anderen kommerziellen Werkzeugen, wie Together/J [Cor04a] oder Microsoft Visio [Cor04c], welche ebenfalls Erweiterungsmöglichkeiten vorsehen, ist der Quell-Code von ArgoUML frei verfügbar. Dies ermöglicht zum einen eine wesentlich gezieltere Anpassung des Werkzeuges an die Erfordernisse, die sich durch BOTL ergebenden, zum anderen ist es hierdurch möglich, den Editor öffentlich zur Verfügung zu stellen.

6.3.1 Funktionsumfang des Editors

Der Editor stellt die Schnittstelle für den Benutzer bei der Erstellung von Metamodellen und Regelwerken dar. Er erlaubt es darüber hinaus, die automatisierte Verifikation der Metamodellkonformität eines Regelwerks anzustoßen und stellt Fehler bei der Verifikation im Regelwerk graphisch dar. Zum Testen von Regelwerken steht ein Demonstrationsmodus zur Verfügung, der es ermöglicht, Quellmodelle graphisch zu modellieren und zu transformieren. Das Ergebnis einer solchen Transformation wird wiederum graphisch in Form eines Objektmodells dargestellt. Im Folgenden werden die wesentlichen Funktionen des Editors kurz skizziert. Weitere Informationen hierzu, sowie ein Tutorial und eine frei verfügbare Version des Werkzeuges finden sich in [Mar04].

Abbildung 6.4 zeigt die Oberfläche des Editors. Für jedes Quellmetamodell, das Zielmetamodell und jede Regel wird im Editor ein eigenes Diagramm angelegt. Auf der linken Seite der Oberfläche befindet sich eine Liste aller aktuell vorhandenen Diagramme. Im unteren Bereich der Oberfläche ist ein Feld zum Editieren der Eigenschaften des aktuell selektierten Diagrammelements vorhanden.

Bestehende BOTL-Projekte können über den Standarddialog Datei → Projekt öffnen geöffnet werden. Im Gegensatz zu ArgoUML-Projekten verfügen diese Projekte über die Dateiendung .botl. Die BOTL-spezifischen Erweiterungen von ArgoUML sind über den Menüeintrag BOTL erreichbar. Abbildung 6.5 zeigt das BOTL-Menu des Editors. Die hier auswählbaren Funktionen sind im Einzelnen:

New BOTL Projekt Ein neues BOTL-Projekt wird angelegt. Es wird jeweils ein leeres Quell- und Zielmetamodell sowie eine leere Regel erzeugt.

New Source Metamodel Ein neues Klassendiagramm wird erzeugt, in dem ein BOTL-Quellmetamodell modelliert werden kann.

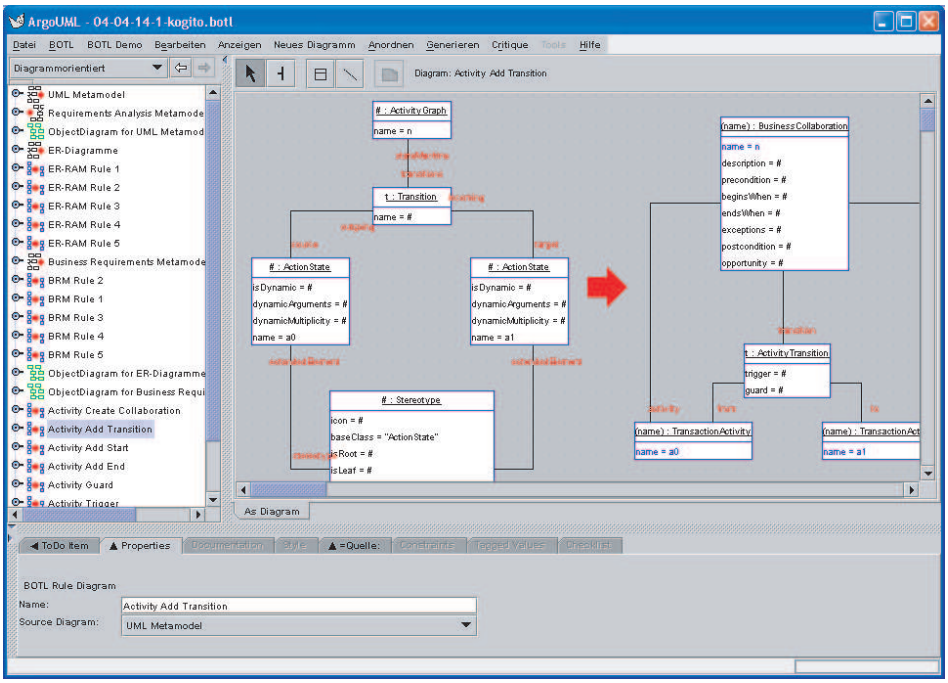


Abbildung 6.4: Oberfläche der ArgoUML-Erweiterung zur Spezifikation von BOTL-Regeln und -Metamodellen

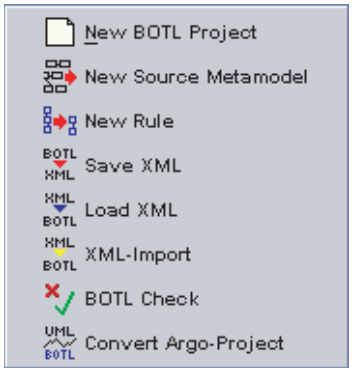


Abbildung 6.5: Menu mit den BOTL-Erweiterungen in ArgoUML

New Rule Ein ein neues Regeldiagramm wird erzeugt, in dem eine BOTL-Regel spezifiziert werden kann. Für jedes Regeldiagramm muss ein Quellmetamodell ausgewählt werden.

Save XML Die aktuelle BOTL-Spezifikation wird als XML-Dokument gespeichert.

Load XML Eine BOTL-Spezifikation in Form eines XML-Dokuments kann selektiert und als neues Projekt geladen werden.

XML Import Eine XML-Dokument mit einer BOTL-Spezifikation wird zusätzlich zu den bereits vorhandenen Regeln und Metamodellen in den Editor importiert. Namensgleiche Regeln und Quellmetamodelle, sowie die Zielmetamodelle werden jeweils zu einem Diagramm zusammengefasst

BOTL Check Die Verifikation der Anwendbarkeit und Metamodellkonformität des aktuellen Regelwerks wird ausgeführt (siehe auch Abschnitt 6.4).

Convert Argo Project Ein ArgoUML-Projekt wird in ein neues BOTL-Projekt konvertiert. In einem nachfolgendem Dialog lässt sich jedes Klassendiagramm des ArgoUML-Projektes wahlweise als Quell- oder Zielmetamodell in das bestehende BOTL-Projekt importieren. Somit lassen sich bestehende Anwendungsdokumentationen als Quell- oder Zielmetamodelle für Transformationen übernehmen.

Abbildung 6.6: Menu mit den BOTL-Erweiterungen für den Demonstrationsmodus

Das in Abbildung 6.6 dargestellte Menü BOTL Demo enthält zwei weitere Einträge für den Demonstrationsmodus:

Create Object Diagrams Für jedes Quellmetamodell des aktuellen Regelwerks wird ein leeres Objektdiagramm angelegt.

Start Transformer Die Quellmodelle werden gemäß dem aktuellen Regelwerk in ein ein Zielmodellfragment transformiert. Das Ergebnis der Transformation wird in einem neu erzeugten Objektdiagramm dargestellt. Dieser Menüeintrag ist nur selektierbar, falls für jedes Quellmetamodell ein Diagramm mit einem Quellmodell existiert.

Für die Spezifikation von Metamodellen werden die vom ArgoUML-Werkzeug unterstützten Klassendiagramme verwendet. Die einzige Erweiterung für die Erstellung von BOTL-Metamodellen ist die Möglichkeit im Eigenschaftsfeld einer Klasse (im unteren Bildbereich) eine Menge von Attributen als Primärschlüssel auszuweisen. In der graphischen Darstellung der Klasse wird das Attribut mit dem Tag <PK> versehen.

Abbildung 6.4 zeigt die Darstellung einer Regel im BOTL-Editor. Objektvariablen und Objektvariablenassoziationen gleichen in ihrer Darstellung UML-Objekten und Links. Das Quellmetamodell einer Regel wird im unteren Eigenschaftsfeld aus der Liste der verfügbaren Metamodelle ausgewählt. Quell- und Zielmodellvariable werden durch einen Pfeil getrennt. Die Zuordnung ob ein Element Teil der Quell- oder Zielmodellvariable erfolgt durch die Auswertung seiner Position relativ zum Pfeil.

Wird eine Objektvariable selektiert, so kann deren Typ aus einer Liste aller Klassen des jeweiligen Metamodells im Eigenschaftsfeld der Objektvariablen ausgewählt werden. Die Attribute und Primärschlüssel der Objektvariablen werden dann automatisch angelegt. Ebenso wird im Eigenschaftsfeld für Objektvariablenassoziationen die Menge möglicher Typen für eine solche Assoziation angeboten. Diese Mechanismen ermöglichen es dem Benutzer auf einfache Weise sicherzustellen, dass die erstellten Regeln konsistent zu den jeweiligen Quell- und Zielmetamodellen sind.

Dieselben Mechanismen kommen auch bei den vom Demonstrationsmodus verwendeten Objektdiagrammen zur Verwendung, um die Konsistenz der Objektmodelle zu ihren Metamodellen weitestgehend zu gewährleisten.

6.3.2 Anbindung der BOTL-Modelle an ArgoUML

Für die Repräsentation von UML-Modellen verwendet ArgoUML die Novosoft UML API (Nsuml) [Nov04] als internes Datenmodell. Diese Bibliothek wird direkt aus der Spezifikation des UML Metamodells in der Version 1.3 generiert. Editoren für die verschiedenen von ArgoUML unterstützten Diagramme sind mit Hilfe des Graphical Edition Framworks (GEF) [Col04b] realisiert. Innerhalb von ArgoUML werden die Elemente der GEF-Modelle auf das zugrundeliegende Nsuml-Modell abgebildet.

Um BOTL-Metamodelle in ArgoUML modellieren zu können, wurden die Klassen für die Darstellung und Bearbeitung von Metamodellen von der Klasse `UMLClassDiagram` des ArgoUML-Werkzeuges abgeleitet. So erben BOTL-Metamodelle innerhalb des Editors alle Eigenschaften von Klassendiagrammen. Somit ist beispielsweise auch eine Codegenerierung aus den Diagrammen, wie sie ArgoUML zur Verfügung stellt, möglich.

Da ArgoUML in der aktuell verwendeten Version 0.12 noch keine Unterstützung für das Zeichnen von Objekten und Links bietet, wurde der für die Modellierung von Regeldiagrammen notwendige Code neu implementiert. Im Gegensatz zu den Klassendiagrammen ist eine Abbildung der Modelle von Regeln in das ArgoUML-eigene UML-Modell nicht möglich. Da ArgoUML die Regeldiagramme jedoch als graphische Information abspeichert, können dennoch alle Diagrammtypen mit dem von ArgoUML zur Verfügung gestellten Speichermechanismus in Form einer BOTL-Projektdatei gesichert werden. Somit bleiben sämtliche Layout-Informationen, Kommentare und zusätzliche graphische Elemente in den Diagrammen erhalten.

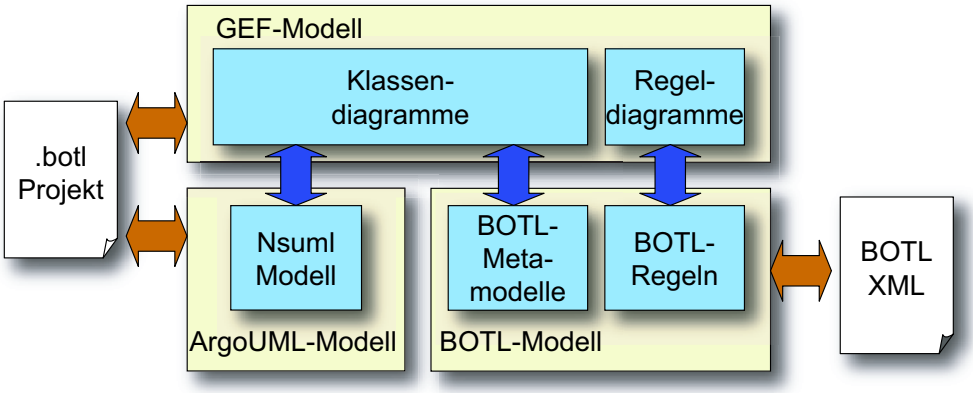


Abbildung 6.7: Zusammenhang zwischen den ArgoUML-Modellen und dem BOTL-Modell

Für die Transformation, den Export und die Verifikation von Regelwerken müssen Regelwerke,

Modelle und Metamodelle in der BOTL-eigenen Representation, also in Form von Instanzen der in Abschnitt 6.2 vorgestellten Klassen, vorliegen. Umgekehrt müssen beim Import von BOTL-XML-Dokumenten in den Editor aus den BOTL-Modellen Diagramme, und somit GEF-Modelle, erzeugt werden. Abbildung 6.7 veranschaulicht den Zusammenhang zwischen GEF und BOTL-Modellen.

Die Klasse `org.argouml.botl.MetaModelBuilder` enthält die notwendigen Methoden für die Abbildung von Regelwerken und Metamodellen auf das GEF-Framework und umgekehrt. Die beiden wesentlichen Methoden hierzu sind:

`public RuleSet buildBOTL()` Diese Methode durchläuft alle Klassen- und Regeldiagramme und erzeugt aus den GEF-Modellen der Diagramme ein BOTL-Modell. Die Methode wird vor jedem XML-Export, bei der Ausführung einer Transformation im Demonstrationsmodus und vor der Verifikation eines Regelwerks aufgerufen.

`public void buildUML(RuleSet rs)` Die Methode erhält ein BOTL-Regelwerk als Aufrufparamter und erzeugt für jede Regel und jedes Metamodell jeweils ein entsprechendes ArgoUML-Diagramm. Sie wird beim Import von BOTL-XML-Dokumenten verwendet.

Im Demonstrationsmodus müssen Quellmodelldiagramme auf BOTL-Quellmodelle und das BOTL-Zielmodell auf die Elemente eines Zielmodelldiagramms abgebildet werden. Diese Abbildungen werden durch die folgenden Methoden der Klasse `org.argouml.BotlDemo` bewerkstelligt:

`public static void initTransformer()` Die Methode liest die Informationen aus den GEF-Modellen sämtlicher Quellmodelldiagramme und erzeugt daraus BOTL-Modellfragmente, die der Transformationskomponente (siehe Abschnitt 6.5) als Eingabe übergeben werden.

`public static void buildDiagram(ModelFragment mf)` Diese Methode erhält beim Aufruf ein Modellfragment als Ergebnis einer Transformation und erzeugt daraus ein Objektdiagramm, das als Zielmodell im Editor dargestellt wird.

Um von der Transformationskomponente verarbeitet werden zu können, müssen die Spezifikationen von BOTL-Transformationen in Form von XML-Dateien vorliegen. Hierzu wird das BOTL-Modell der Regeln und ihrer Metamodelle als XML-Dokument exportiert. Beim Laden von XML-Spezifikationen wird intern ein BOTL-Modell aufgebaut, aus dem wiederum GEF-Modelle für die Regel- und Klassendiagramme erzeugt werden.

Für den Export und Import der XML-Darstellungen eines BOTL-Regelwerks wird das Castor Datenanbindungs-Framework verwendet [EG04]. Castor ist ein Open-Source-Framework, welches es erlaubt Java-Objektgeflechte als XML-Dokumente zu exportieren und umgekehrt XML-Dokumente in Java-Objektstrukturen zu importieren. Die XML-Anbindung ist im Rahmen eines studentischen Systementwicklungsprojekts entstanden und in [Tod03] detailliert beschrieben.

6.4 Verifikation von Regelwerken

Innerhalb des Java-Pakets `de.tum.in.botl.heuristics` finden sich alle notwendigen Klassen für den Nachweis, der in Kapitel 4 eingeführten Eigenschaften der Anwendbarkeit und der Metamodellkonformität eines Regelwerks. Das Werkzeug ArgoUML wurde so erweitert, dass diese Überprüfung vom BOTL-Editor aus angestoßen werden und das Ergebnis der Überprüfung graphisch dargestellt werden kann. Die grundlegende Infrastruktur, sowie Ansätze zur Implementierung der Verifikationstechniken aus Kapitel 4 wurden im Rahmen eines Systementwicklungsprojektes erarbeitet

[Tra04]. Die wesentlichen, durch diese Komponente und die Erweiterungen von ArgoUML realisierten Funktionalitäten, sind im Einzelnen:

- Verifikation der Anwendbarkeit eines Regelwerks
- Verifikation der Metamodellkonformität eines Regelwerks
- Erstellung eines für den Anwender nachvollziehbaren Protokolls der Verifikation eines Regelwerks
- Darstellung von im Verlauf der Verifikation gefundenen Fehlern in Form von Kommentaren im BOTL-Editor

Die Umsetzung der einzelnen Teilfunktionalitäten wird in den nachstehenden Unterabschnitten kurz diskutiert.

6.4.1 Verifikation der Anwendbarkeit von Regelwerken

Die Verifikation der Anwendbarkeit von Regelwerken erfolgt durch die Methoden der Klasse

`de.tum.in.botl.heuristics.implementation.Applicability`.

Die Methode

```
public static boolean isApplicable(RuleSet rs)
```

erhält ein BOTL-Regelwerk als Aufrufparameter und liefert den Boolschen Wert `true` zurück, falls die Verifikation der Anwendbarkeit des Regelwerks erfolgreich verläuft. Der Nachweis erfolgt im Wesentlichen in zwei Schritten: Zunächst wird versucht, die Anwendbarkeit jeder einzelnen Regel gemäß Definition 4.1.6 zu verifizieren. In einem zweiten Schritt wird sichergestellt, dass sämtliche Zielobjektvariablen des Regelwerks konfliktfrei sind. Die Anwendbarkeit einer einzelnen Regel wird durch die Methode

```
public static boolean isApplicable(Rule r)
```

verifiziert. Innerhalb dieser Methode wird eine Reihe weiterer Methoden verwendet, um die syntaktische Korrektheit der Regel zu überprüfen und mit Hilfe der Verifikationstechniken aus Abschnitt 4.1 die einzelnen Kriterien für die Anwendbarkeit des Regelwerks nachzuweisen. In der aktuellen Version der Verifikationskomponente existiert jedoch noch keine Möglichkeit Systeme aus Gleichungen und Ungleichungen zu lösen. Daher beschränkt sich die Verifikation der Anwendbarkeit einer Regel an diesen Stellen auf einfachere Techniken, die ohne einen solchen Mechanismus realisierbar sind. Für den Nachweis der Anwendbarkeit einer Regel werden die folgenden Methoden verwendet:

`boolean checkSyntax(Rule r)` überprüft die synaktische Korrektheit einer Regel. Die Korrektheit der Syntax von Regeln wird bereits weitestgehend durch den Editor sichergestellt. Daher beschränkt sich die Überprüfung an dieser Stelle darauf, zu verifizieren, dass kein Primärschlüsselattribut einer Zielobjektvariable den Wert $\diamondsuit$ aufweist. In diesem Fall würde das Gleichungssystem *GL* aus Definition 3.5.4 keine eindeutig Lösung liefern und die Regel wäre somit nicht anwendbar.

`boolean createsValidFragments(Rule r)` überprüft, ob eine Regel ausschließlich gültige Modellfragmente gemäß Definition 4.1.3 erzeugt. Da derzeit noch keine Möglichkeit zum Auswerten von Ungleichungssystemen besteht, wird die in Satz 4.1.2 eingeführte, einfache Verifikationstechnik für *createsValidFragments* verwendet.
Die Methode `solveSystem(Rule r)` überprüft hierzu, ob das Gleichungssystem *GL* der Regel höchstens eine Lösung hat (Satz 4.1.2 (i)). Dies ist dann nicht der Fall, wenn ein Term in der Zielmodellvariablen eine Variable enthält, die auf der linken Regelseite nicht vorkommt. Lässt sich für einen Identifikator oder ein Attribut der Zielmodellvariablen kein Wert bestimmen, so wird ein Hinweis ausgegeben, dass die Regel Constraint-behaftet ist. Eine weitere Auswertung der Constraints erfolgt an dieser Stelle nicht, da beliebige, nicht erfüllte Constraints gemäß Definition 3.5.5 immer zu einem leeren Modellfragment als Ergebnis führen. Demzufolge können Regeln auch über Constraints verfügen, die ggf. niemals erfüllt sind.
Mit Hilfe der Methode `similarOV(ObjectVariable ov1, ObjectVariable ov2)`, wird schließlich für alle Paare von Zielmodellvariablen der Regel kontrolliert, ob die rechte Regelseite ähnliche Objektvariablen gemäß Definition 3.5.8 enthält. Ist dies nicht der Fall, so ist auch Forderung (ii) aus Satz 4.1.2 erfüllt.

`boolean deterministic(Rule r)` überprüft für jede Zielmodellvariable einer Regel, ob diese deterministisch gemäß Definition 4.1.5 ist. Hierzu wird die in Satz 4.1.4 vorgestellte Verifikationstechnik verwendet. Dementsprechend wird für jede Zielobjektvariable *ov* geprüft, ob eines der folgenden drei Kriterien gilt:

- Satz 4.1.4 (i): Der Identifikator der Objektvariablen hat den Wert $\diamondsuit$.
- Satz 4.1.4 (ii): Alle Attribute sind entweder Konstante oder haben den Wert $\diamondsuit$. Diese Eigenschaft wir durch die Methode
  `areAllAVsDiamondsOrConstants( ObjectVariable ov)` überprüft.
- Satz 4.1.4 (iii): Für alle Attribute der Objektvariablen gilt:

$$dependsOn(ov,r) \overset{r|_{mv_0}}{\rightsquigarrow} attDependsOn(ov,a,r)$$

  Für den Nachweis dieser Eigenschaft stehen drei Hilfsmethoden zur Verfügung:
  `List dependsOn(ObjectVariable ov, Rule r)` realisiert die Abbildung *dependsOn* aus Definition 4.1.6. Mittels `getVarNames()` werden sämtliche Variablen des Objektvariablenidentifikators, bzw. aller Primärschlüsselattribute, falls solche existieren, ermittelt. Als Ergebnis wird eine Liste sämtlicher Objektvariablen der linken Regelseite, welche zumindest eine dieser Variablen enthalten, übergeben.
  `List attDependsOn(ObjectVariable ov, Attribute a, Rule r)` realisiert die Abbildung *attDependsOn* aus Definition 4.1.7. Sie ist in der gleichen Weise wie die Methode `dependsOn()` verwirklicht.
  `boolean determines(ModelVariable mv, List l1, List l2)` realisiert die Relation $\overset{mv}{\rightsquigarrow}$ aus Definition 4.1.8. Die Gültigkeit der Relation wird anhand von Satz 4.1.3 nachgewiesen. Dementsprechend überprüft die Methode, ob in der Modellvariablen `mv` zu jeder Objektvariablen der Liste `l2` ein Pfad von Objektvariablen der Liste `l1` existiert. Die Relation gilt, falls entlang dieses Pfades die Typen aller eingehenden Objektvariablenassoziationen mit der Multiplizitätsobergrenze 1 versehen sind. Die Methode ruft für jede Objektvariable der zweiten Liste eine Hilfsmethode auf,

welche rekursiv nach allen möglichen Pfaden zu dieser Objektvariablen in der gegeben Modellvariablen sucht.

Ist eines dieser drei Kriterien erfüllt, so liefert die Methode `deterministic` den Wert `true`, andernfalls wird `false` zurückgegeben.

`boolean conflictFree(Rule r)` kontrolliert für sämtliche Paare von Zielobjektvariablen einer Regel, ob diese konfliktfrei gemäß Definition 4.1.9 sind. Hierzu wird die einfache Verifikationstechnik für den Nachweis der Konfliktfreiheit zweier Objektvariablen aus Lemma 4.1.6 verwendet.

Zunächst wird durch den Aufruf der Methode `similarOV()` überprüft, ob zwei Objektvariablen konfliktgefährdet sind (Lemma 4.1.6 (4.19)). Ist dies der Fall, so vergleicht die Methode

```
compareAttributes( ObjectVariable ov1,
                   ObjectVariable ov2 )
```

sämtliche Attributwerte der beiden Objektvariablen gemäß Lemma 4.1.6 (4.20). Erweisen sich die beiden Objektvariablen als konfliktfrei, so wird der Wert `true` zurückgeliefert.

Um die Anwendbarkeit des gesamten Regelwerks sicherzustellen, wird im Anschluss an die Überprüfung der einzelnen Regeln gemäß Definition 4.1.7 (ii) überprüft, ob sämtliche Zielobjektvariablen aus unterschiedlichen Regeln des Regelwerks konfliktfrei sind. Hierzu wird die Methode `boolean conflictFree(RuleSet rs)` verwendet. Zielobjektvariablen aus derselben Regel werden von dieser Methode nicht verglichen, da diese Überprüfung bereits bei der Verifikation der Anwendbarkeit der einzelnen Regeln durchgeführt wurde.

6.4.2 Verifikation der Metamodellkonformität von Regelwerken

Um die Metamodellkonformität eines Regelwerks zu verifizieren werden die Methoden der Klasse

```
de.tum.in.botl.heuristics.implementation.Conformance
```

verwendet. Die Klasse verfügt über zwei wichtige, öffentlich zugängliche Methoden:

`public static boolean ubConform(RuleSet rs)` überprüft die Obergrenzenkonformität des Regelwerks.

`public static boolean lbConform(Rule r)` überprüft die Untergrenzenkonformität einer einzelnen Regel.

Die folgenden Abschnitte gehen kurz auf die technische Umsetzung dieser beiden Methoden ein.

Verifikation der Obergrenzenkonformität

Für den Nachweis der Obergrenzenkonformität gilt es zunächst Mengen von Objektvariablen, die potentiell dasselbe Objekt erzeugen, zu bestimmen (Abschnitt 4.2.2, S. 141 ff). Weiterhin muss eine Heuristik für die Funktion *maxmatch* (Abschnitt 4.2.2, S. 146 ff) realisiert werden, auf deren Basis sich wiederum eine Implementierung der Funktion *ubVarCard* verwirklichen lässt. Diese wird in der hier vorliegenden Werkzeugunterstützung verwendet, um die Obergrenzenkonformität eines Regelwerks mit Hilfe von Satz 4.2.3 nachzuweisen.

Da die Identifizierung redundanter Objektvariablenassoziationen nur für eine Reihe von Spezialfällen von echter Relevanz ist, wurde sie in der vorliegenden Version des Werkzeuges noch nicht realisiert. Im Bedarfsfall kann die Redundanz von Objektvariablenassoziationen jedoch mit den in Abschnitt 4.2.2 (S. 135 ff) vorgestellten Techniken manuell nachgewiesen werden.

Um Mengen von Objektvariablen zu identifizieren, die potentiell dasselbe Objekt im Zielmodell erzeugen, wird durch die Methode

```
private static java.util.Hashtable getOVConf(RuleSet rs)
```

die Menge $\mathbb{OV}_R^{conf}$ (s. Definition 4.2.17) gebildet.

Die Methode liefert für ein Regelwerk eine `HashTable`, deren Schlüssel jeweils die Menge aller Klassen der Zielmetamodelle sind. Als Wert erhält man für jede Klasse ein Objekt des Typs `Vector`, dessen Einträge wiederum aus Mengen von Objektvariablen bestehen, die potentiell dasselbe Objekt im Zielmodell erzeugen können. Um solche Objektvariablen identifizieren zu können, stützt sich die Implementierung von `getOVConf()` auf eine eingeschränkte Version der Relation *cannotCreateSame* (Definition 4.2.15), die nur die beiden Forderungen (i) und (ii) der Definition berücksichtigt. Dementsprechend werden mehr Objektvariablen als unbedingt notwendig in einer Klasse von Objektvariablen zusammengefasst. Hierdurch ist zwar der Nachweis der Obergrenzenkonformität für eine Menge obergrenzenkonformer Regelwerke nicht mehr möglich, die Korrektheit eines Nachweises dieser Eigenschaft bleibt jedoch erhalten, falls ein solcher Nachweis möglich ist.

Forderung (i) der Definition, welche besagt, dass die Objektvariablen Zielobjektvariablen desselben Regelwerks sein müssen, ist durch die interne Repräsentation von Regelwerken im Werkzeug ohnehin sichergestellt (siehe Abschnitt 6.2). Aussage (ii) besagt, dass unähnliche Objektvariablen keine gleichen Objekte erzeugen können. Für den Nachweis dieser Eigenschaft wird die bereits in Abschnitt 6.4.1 vorgestellte Methode `similarOV()` verwendet.

Die Funktion *maxmatch* liefert gemäß Definition 4.2.18 (S. 147) für eine Quellmodellvariable die höchstmögliche Anzahl von Matches in einem beliebigem Quellmodell, bei der eine Menge $M1$ ihrer Objektvariablen immer paarweise auf dieselben Objekte matcht, während eine Menge $M2$ von Objektvariablen höchstens einmal auf dieselbe Menge von Objekten matcht. Die Funktion wird innerhalb des Werkzeuges durch die Methode

```
private static int maxmatch( ModelVariable mv,
                             List m1, List m2 )
```

realisiert. Derzeit existiert lediglich eine Implementierung der Methode `maxmatch`, die für eine eingeschränkte Menge von Sonderfällen korrekt arbeitet. Sie erlaubt nur dann die Berechnung eines Wertes für *maxmatch*, falls die entsprechende Modellvariable zykelfrei und die Menge `m1` höchstens einelementig ist, was beim Aufruf der Methode durch die Hilfsfunktion `findCycle()` überprüft wird.

Die Methode

```
private static in ubVarCard( ObjectVariableAssociation ova,
                             ClassAssociationEnd ae1,
                             Rule r )
```

realisiert die in Definition 4.2.21 (S. 153) eingeführte Funktion *ubVarCard*. Die Methode stützt sich auf die bereits vorgestellte Methode `dependsOn()` und verwendet in Abhängigkeit ihres Ergebnisses (siehe Tabelle 4.2, S. 154) ggf. die Methode `maxmatch`, um das Ergebnis von *ubVarCard* zu ermitteln.

Zum Nachweis der Obergrenzenkonformität eines Regelwerks wird nun gemäß Satz 4.2.3 für jedes Klassenassziationsende die Summe aller *ubVarCard*-Werte ermittelt, wobei sämtliche Objektvariablenassoziationen an Objektvariablen aus jeweils derselben Menge aus dem Ergebnis von `getOVConf()` stammen. Ist das Maximum über alle von `getOVConf()` ermittelten Mengen von Objektvariablen für ein Klassenassoziationsende kleiner als dessen Multiplizitätsobergrenze, so ist die in Satz 4.2.3 (i) geforderte Eigenschaft für dieses Ende erfüllt. Ist die Eigenschaft für sämtliche Klassenassoziationsenden des Zielmetamodells erfüllt und konnte die Anwendbarkeit des Regelwerks bereits nachgewiesen werden (Forderung (ii) in Satz 4.2.3), so liefert `ubConform(RuleSet rs)` den Wert `true`. Andernfalls wird `false` als Ergebnis des Methodenaufrufs zurückgegeben.

Verifikation der Untergrenzenkonformität

Zur Verifikation der Untergrenzenkonformität einer Regel wird zunächst versucht, den Nachweis anhand der in Lemma 4.2.4 vorgestellten, einfachen Verifikationstechnik für Untergrenzenkonformität zu erbringen. Hierfür muss gezeigt werden, dass für die Zielmodellvariable *mv* einer gegeben Regel das Prädikat $varLbConform(mv)$ gilt. Dieses besagt im Wesentlichen, dass alle Objektvariablen des Zielmodellfragments bereits die nötige Anzahl ausgehender Assoziationen eines Typs aufweisen. Die Überprüfung dieser Eigenschaft erfolgt durch die Methode

```
private static boolean varLbConform( ModelVariable mv ).
```

Die Methode durchläuft sämtliche Klassenassoziationsenden des Metamodells der Modellvariablen. Für jede Objektvariable, deren Typ zum jeweiligen Klassenassoziationsende passt, wird überprüft, ob die Summe der ausgehenden Assoziationen dieses Typ mindestens so groß ist wie die im gegenüberliegenden Assoziationsende geforderte Multiplizitätsuntergrenze.

Ist der Nachweis der Untergrenzenkonformität für eine Regel mit der einfachen Verifikationstechnik nicht möglich, so wird die erweiterte Technik aus Satz 4.2.6 hierfür verwendet. Die Implementierung stützt sich hierbei auf die beiden Hilfsmethoden

```
private static int minmatch(ModelVariable mv,
                            List m1, List m2 )
```

und

```
private static int lbVarCard(ObjectVariableAssociation ova,
                             ClassAssociationEnd ae1,
                             Rule r ).
```

Diese sind vollkommen analog zu den Methoden für den Nachweis der Obergrenzenkonformität realisiert und werden daher an dieser Stelle nicht eingehender diskutiert. Kann die Untergrenzenkonformität einer Regel mit einer der beiden Techniken nachgewiesen werden, so liefert die Methode `lbConform()` das Ergebnis `true`, andernfalls wird `false` als Ergebnis zurückgegeben.

6.4.3 Erstellung eines Verifikationsportokolls und Darstellung von Fehlern

Ein wesentliches Ziel der Verifikation der Metamodellkonformität eines Regelwerks ist es, das Ergebnis der Überprüfung in einer für den Benutzer verständlichen und nachvollziehbaren Form zu dokumentieren. Aus diesem Grund wird im Verlauf der Verifikation ein textueller Beweis der gewünschten Eigenschaften generiert, soweit dieser Nachweis erbringbar ist. Weiterhin werden Fehler,

die dazu führen, dass das gegebene Regelwerk nicht metamodellkonform ist, graphisch im Regeleditor angezeigt. Die wesentlichen Klassen hierfür finden sich in dem in Abbildung 6.8 dargestellten Paket `tum.in.botl.errorProcessing`.

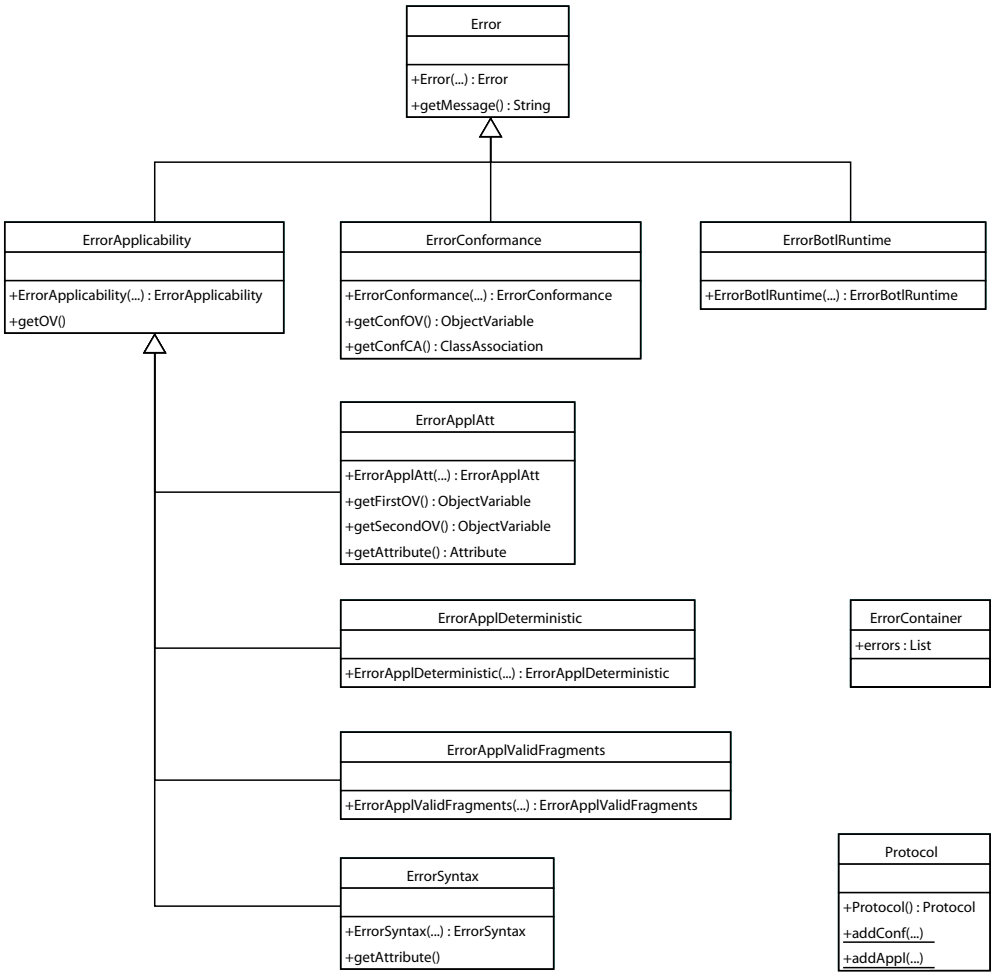


Abbildung 6.8: Das Paket `errorProcessing` mit den Klassen für die Behandlung von Fehlern während der Verifikation eines Regelwerks

Für den Nachweis der Anwendbarkeit und der Metamodellkonformität wird jeweils ein eigenes Protokoll erstellt, dessen Inhalt in der statischen Klasse

`de.tum.in.botl.excProccessing.implementation.Protocol`

gekapselt ist. Mittels der Methoden `addConf(String s, int tab)` und `addAppl(String s, int tab)` wird je eine um den Wert von `tab` eingerückte Textnachricht an das entsprechende Protokoll angefügt. Im Verlauf der Verifikation eines Regelwerks schreibt jede aufgerufene Methode Meldungen über das Ergebnis ihres Aufrufs in das jeweilige Protokoll. Abbildung 6.9 zeigt ein Beispiel für eine Ausgabe nach der Verifikation eines Regelwerks.

Schlägt ein Teil der Verifikation eines Regelwerks fehl, so wird jeweils ein Fehlerobjekt erzeugt, in dem Art und Ursache des Fehlers dokumentiert werden. Je nach Art des Fehlers werden dem Fehler-

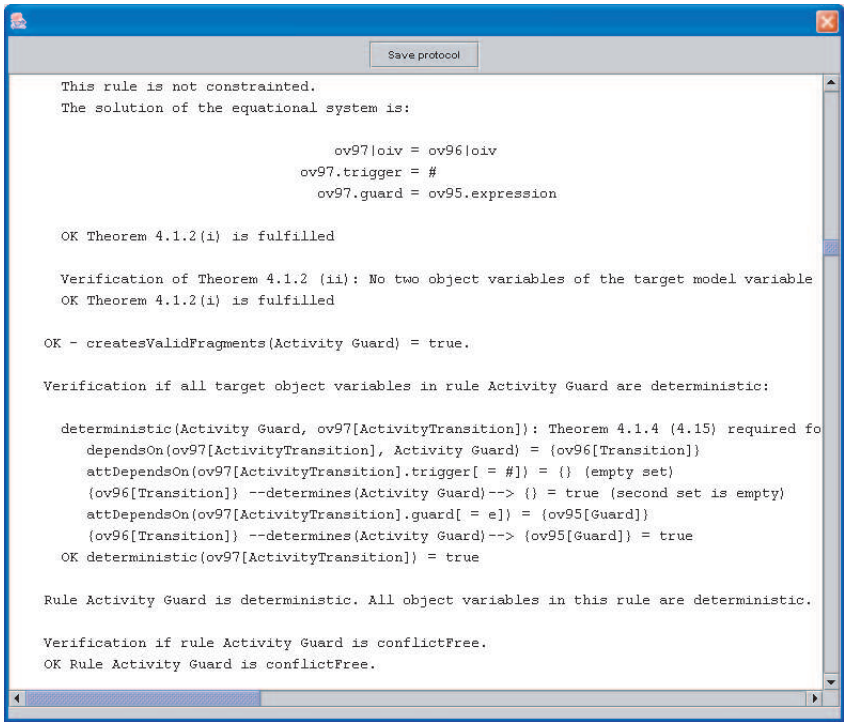


Abbildung 6.9: Ausgabe des Protokolls nach der Verifikation eines Regelwerks

objekt bei seiner Erzeugung ggf. eine oder mehrere Objektvariable, sowie Verweise auf Attribute der Objektvariablen, als Parameter mitgeliefert. Erweist sich ein Regelwerk als nicht metamodellkonform, so wird die Verifikation dennoch fortgeführt und die auftretenden Fehler in einer statischen Liste der Klasse `errorContainer` gesammelt.

Der Aufruf der Verifikationskomponente erfolgt innerhalb von ArgoUML in der Klasse `org.argouml.BOTL.RuleSetVerificator`. Der Vorgang wird durch die den Aufruf der Methode

```
void checkBOTLRuleSet(RuleSet rs, Hashtable botl2gef)
```

gestartet. Diese erhält als Parameter das zu überprüfende Regelwerk und eine Tabelle, in welcher die Zuordnung von Elementen des BOTL-Modells zu den GEF-Elementen der Diagramme enthalten ist. Diese Tabelle wird benötigt, um die Elemente in Regeldiagrammen aufzufinden, an denen Fehler angezeigt werden sollen. Nach Ablauf des Verifikationsvorgangs wertet der Regeleditor die gesammelten Fehlerobjekte aus und stellt sie in Form von Kommentaren mit Hinweisen auf die Art des Fehlers an den entsprechenden Elementen eines Regelwerks dar.

Ein Beispiel für eine Regel, deren Objektvariable mit entsprechenden Hinweisen versehen wurde, ist in Abbildung 6.10 dargestellt. Betroffene Attribute einer Objektvariablen werden jeweils rot eingefärbt.

6.5 Die Transformationskomponente

Das Paket `de.tum.in.botl.transformer` beinhaltet die Komponente zur Transformation von Objektmodellen. Die Transformationskomponente ist eine eigenständige Java-Komponente, die unab-

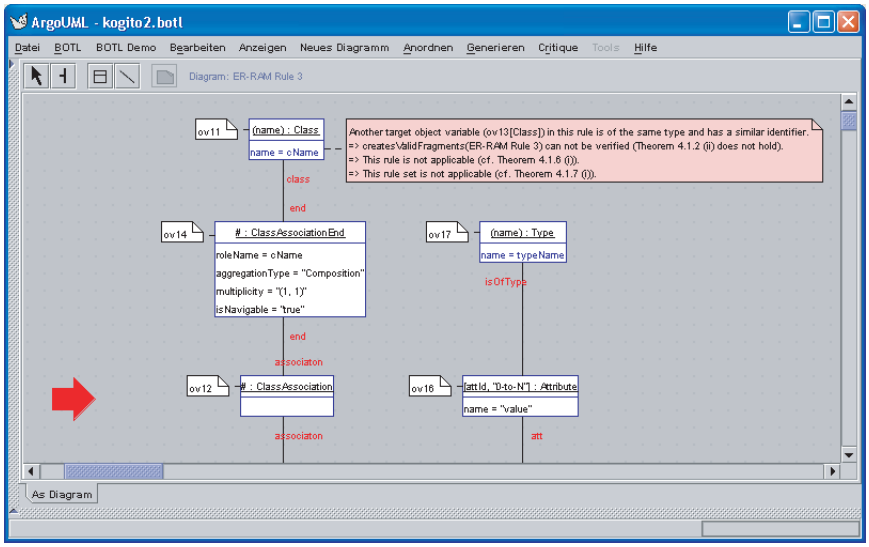


Abbildung 6.10: Ausgabe von Fehlermeldungen nach der Verifikation eines Regelwerks

hängig von der ArgoUML-Anwendung betrieben werden kann. Sie transformiert BOTL-spezifische Repräsentationen von Objektmodellen gemäß der in Abschnitt 3.5 eingeführten Semantik für BOTL-Regelwerke.

Als Eingabe benötigt sie die von dem in Abschnitt 6.3 vorgestelltem BOTL-Editor erzeugte XML-Darstellung eines Regelwerks und dessen Quell- und Zielmetamodellen. Um reale Objektmodelle in technischen Formaten, wie z.B. Java-Objektstrukturen, zu transformieren, werden diese mit Hilfe von Importern in die interne BOTL-Darstellung umgewandelt und das Ergebnis mit Hilfe eines Exporters wieder in das gewünschte Ausgabeformat gebracht. Im Rahmen dieser Arbeit entstand hierzu ein Importer und ein Exporter, die es erlauben, Java-Objektgeflechte einzulesen und auszugeben.

Der Ablauf bei der Ausführung einer Modelltransformation durch die hier vorgestellte Komponente ist in Abbildung 6.11 in Form eines UML-Aktivitätsdiagramms dargestellt. Zunächst werden die zu transformierenden Quellmodelle in den Transformator importiert. Nun werden nacheinander sämtliche Regeln des betreffenden Regelwerks abgearbeitet.

Für jede Regel werden zunächst Matches der Quellmodellvariablen im im jeweiligen Quellmodell gesucht. Ist die Regel Constraint-behaftet (z.B. durch eine Variable die in verschiedenen Attributvariablen vorkommt), so wir überprüft ob sämtliche Constraints durch den gefundenen Match erfüllt sind. Ist dies nicht der Fall so wird der betreffende Match verworfen.

Wurden sämtliche gültige Matches Quellmodell der aktuellen Regel gefunden, so wird für jeden dieser Matches ein Zielmodellfragment entsprechend der Struktur der Zielobjektvariablen erzeugt. Für jedes erzeugte Modellfragment werden Werte für die Attribute und Identifikatoren seiner Objekte als Lösungen des Gleichungssystems *GL* (siehe Definition 3.5.4) errechnet und das neu erzeugte Modellfragment wird in das Zielmodell eingebettet.

Nach Abarbeitung aller gefunden Matches für alle Regeln werden noch nicht belegte Attributwerte mit Default-Werten gefüllt. Abschließend wird das so erzeugte Zielmodellfragment in dem jeweils gewünschten, technologiespezifischen Ausgabeformat exportiert.

Innerhalb der nachstehenden Teilabschnitte wird die Realisierung der wichtigsten Aktivitäten im Verlauf einer Modelltransformation durch die Transformationskomponente kurz vorgestellt.

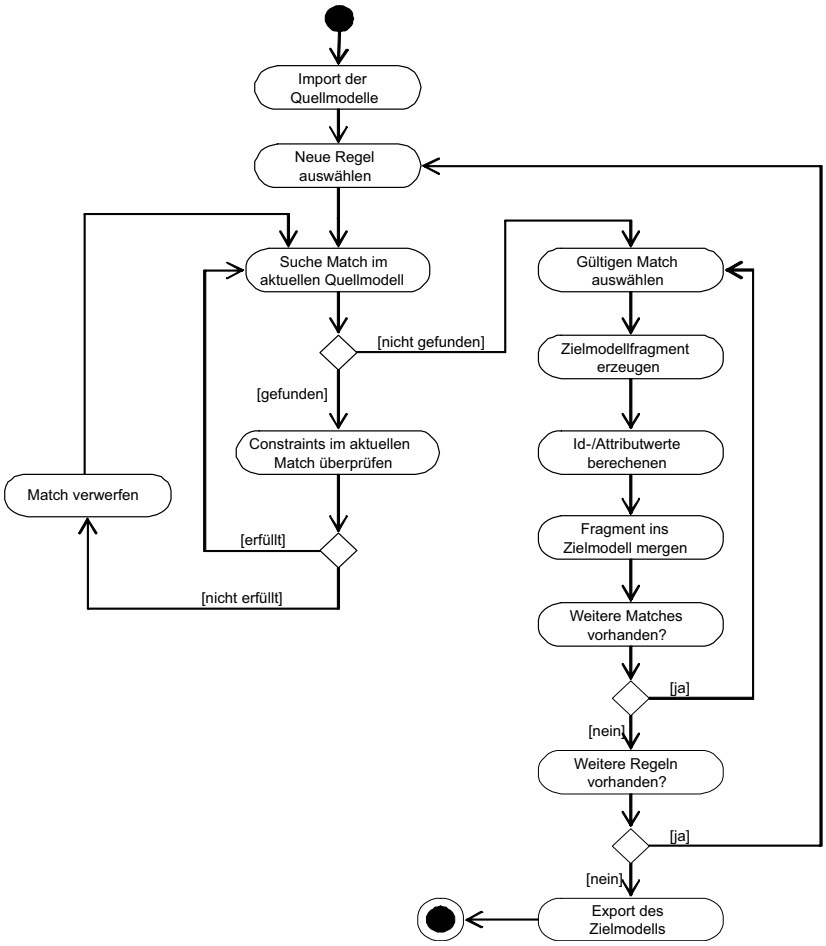


Abbildung 6.11: Ablauf einer Modelltransformation

6.5.1 Lösen von Gleichungen und Gleichungssystemen

Das (symbolische) Lösen von Gleichungssystemen ist im Verlauf einer Transformation an mehreren Stellen erforderlich. Innerhalb dieses Abschnitts wird daher kurz skizziert, wie diese Aufgabe innerhalb des BOTL-Werkzeuges erfüllt wird. Da für das Werkzeug ausschließlich frei verfügbare Komponenten verwendet werden sollen, ist eine Integration kommerzieller Anwendungen wie Mathematica [WR04] oder MatLab [TM04] nicht möglich. Statt dessen wird eine Reihe von frei verfügbaren Werkzeugen kombiniert, um Gleichungen und Gleichungssysteme symbolisch lösen zu können.

Lösen von Gleichungen

Klassen zum symbolischen Lösen von Gleichungen müssen das Interface

```
de.tum.in.botl.transformer.interfaces.EquationSolverInterface
```

implementieren. Dies erlaubt es, an dieser Stelle ggf. leistungsfähigere, kommerzielle Produkte einzusetzen, indem diese entsprechend gekapselt werden. Im Rahmen der hier erstellten Werkzeugunterstützung wird dieses Interface von der Klasse

```
de.tum.in.botl.transformer.Implementation.EquationSolver
```

implementiert. Diese Klasse kapselt das frei verfügbare Java-Paket `SymJPack`, das von Jens-Uwe Dolinsky entwickelt wurde. `SymJPack` ist ein Computeralgebrasystem, welches symbolisches Rechnen ermöglicht. Neben der symbolischen Auswertung eines Terms ist es auch möglich, eine Gleichung nach einer gegebenen Variable symbolisch zu lösen.

Zum Lösen einer Gleichung wird die Methode

```
public String solveEquation( String term,
                             String variable )
```

verwendet, die einen Term, der den Wert Null liefern soll, und die unbekannte Variable, nach der der Term aufgelöst werden soll, als Parameter des Typs `String` erhält. Als Ergebnis liefert die Methode einen String, der den entsprechenden Ergebnisterm enthält. Beispielsweise liefert der Aufruf

```
solveEquation("x + 2x - a + 3", "x");
```

den String `"1 - a/3"` als Ergebnis.

Die Möglichkeit, einzelne Gleichungen symbolisch nach einer Unbekannten auflösen zu können, ist eine Voraussetzung für die im folgenden Abschnitt vorgestellte Methode zum symbolischen Lösen von Gleichungssystemen.

Lösen von Gleichungssystemen

Das Interface

```
de.tum.in.botl.transformer.interfaces
                      .EquationalSystemSolverInterface
```

muss von allen Klassen, die für das Lösen von Gleichungssystemen verwendet werden, implementiert werden. Wie auch beim symbolischen Lösen von Gleichungen kann die existierende Implementierung des Gleichungssystemlösers ggf. durch eine leistungsstärkere, kommerzielle Variante ersetzt werden. Innerhalb des Prototypen wird dieses Interface von der Klasse

```
de.tum.in.botl.transformer.implementation
                    .EquationalSystemSolver
```

implementiert. Ein Gleichungssystem wird durch den Aufruf der statischen Methode

```
public static Hashtable solveSystem( Vector terms,
                                     Vector vars  )
```

gelöst. Diese erhält als Parameter zwei Vektoren aus Strings. Der erste Vektor enthält eine Menge von Termen für, die sich jeweils der Wert Null ergeben soll, und der Zweite die Menge der unbekannten Variablen in diesen Termen. Als Ergebnis liefert die Methode eine Tabelle vom Typ `Hashtable`, in der jeder als unbekannt deklarierten Variable ein String mit der entsprechenden Lösung zugeordnet ist.

Die Klasse `EquationalSystemSolver` verwendet das unter GNU-Lizenz entwickelte Java-Paket `jscl-meditor` [JS04], um Gleichungssysteme symbolisch zu lösen. `jscl-meditor` beinhaltet eine Bibliothek für symbolische Algebra in Java, zusammen mit einer, im Rahmen dieser Arbeit nicht verwendeten, graphischen Oberfläche. Die wesentliche Eigenschaft der Java-Bibliothek ist die Möglichkeit, ein Gleichungssystem in eine Gröbner-Normalform [AP94, BW93, Amr94] umzuwandeln. Ist eine solche Normalform für ein Gleichungssystem möglich, so besteht das Gleichungssystem aus einer Reihe von Gleichungen, von denen die erste lediglich eine Unbekannte, die zweite dieselbe Unbekannte und eine weitere, usw. enthält.

Mit Hilfe der Klasse `EquationSolver` wird nun die erste Gleichung nach der ersten Unbekannten symbolisch gelöst. In allen verbleibenden Gröbner-Gleichungen wird die Unbekannte durch die Lösung substituiert und der Vorgang solange sukzessive fortgesetzt, bis für jede Unbekannte eine Lösung berechnet wurde. Für eine Unbekannte *var* die nicht im Gleichungssystem auftaucht wird das Ergebnis

$$var = var$$

zurückgeliefert. Ist eine eindeutige Berechnung von Lösungen für alle Unbekannte nicht möglich, so wird eine Ausnahme vom Typ `TransformationException` mit den entsprechenden Fehlerinformationen geworfen.

Betrachtet wird das folgende exemplarische Codefragment:

```
Vector terms = new Vector();
Vector vars = new Vector();

terms.addElement("a-x^2+y");
terms.addElement("(b-2*x+3)");
terms.addElement("2*x+4*z");

vars.addElement("x");
vars.addElement("y");
vars.addElement("z");

Hashtable result =
      EquationalSystemSolver.solveSystem(terms, vars);
```

Nach dem Aufruf der `solveSystem`-Methode enthält die Hashtabelle `result` die folgenden Werte:

```
x = (3+b)/2
z = -((b+3)/4)
y = -(((((4*a)-9)-(6*b))-b^2)/4)
```

6.5.2 Import von Java-Objektgeflechten

Zur Transformation von Modellen müssen diese zunächst in die werkzeuginterne Darstellung, in Form von Instanzen der Klassen aus dem Paket `de.tum.in.botl.model` gebracht werden. Für verschiedene technische Darstellungsformen von Modellen werden jeweils spezielle Importer benötigt, die es erlauben, die einzelnen Modelle in die einheitliche BOTL-interne Darstellung zu transformieren. Dies erlaubt auch die Kombination unterschiedlicher Formate von Quell- und Zielmodellen. So ist es bei der Verwendung mehrerer entsprechender Adapter beispeilsweise möglich, Java- und CORBA-Objektgeflechte zusammen mit XMI-Dokumenten in ein beliebiges Zielmodell zu transformieren.

Ein BOTL-Importer muss das Interface

```
de.tum.in.botl.adapter.interfaces.Importer
```

implementieren. Dieses sieht im wesentlichen zwei Methoden vor:

`void initialize(Metamodel mm)` Diese Methode initialisiert den Importer, indem sie angibt, welche Art von Modellen importiert werden soll. Hierzu wird im Parameter `mm` ein Metamodell mit übergeben.

`ModelFragment start(Object obj)` Die `start`-Methode erhält eine Referenz auf ein Objekt einer Objektstruktur, welche vom Importer anschließend rekursiv durchsucht wird. Der Inhalt der Objektstruktur wird in ein BOTL-Modellfragment konvertiert und als Ergebnis des Imports zurückgeliefert.

In der vorliegenden Arbeit wurde prototypisch ein Importer für Java-Objektgeflechte realisiert. Dieser verwendet das Java-Reflection Framework, um zur Laufzeit den Typ von Java-Objekten festzustellen und Informationen über die Attribute und Assoziationen der Java-Klassen zu ermitteln.

Der Vorteil dieser Lösung liegt darin, dass der Java-Importer sehr vielseitig einsetzbar ist. Für seine Verwendung werden keine Informationen über die Klassenstruktur der zu importierenden Daten benötigt. Allerdings müssen sämtliche Attribute von zu importierenden Java-Klassen als öffentlich (`public`) deklariert sein, um der Reflection-API einen Zugriff auf die Objekt-Attribute zur Laufzeit zu ermöglichen. Alternativ ließen sich Importer aus dem Java-Code der anzubindenden Anwendung generieren. Hierbei bestünde dann die Möglichkeit explizit anzugeben, welche Getter-Methoden verwendet werden müssen, um auf Attributwerte von Objekten zugreifen zu können.

Der Importer der Referenzimplentierung durchläuft rekursiv das ihm übergebene Objektgeflecht und baut dabei ein BOTL-Modellfragment auf. Damit ein Java-Objekt importiert wird, muss sein Klassennamen im BOTL-Metamodell bekannt sein. Objekte, Attribute und Assoziationen die nicht im BOTL-Metamodell vorkommen werden beim Import ignoriert. Dies ermöglicht den Import von Java-Objektstrukturen, deren Klassen um zusätzliche, ggf. technisch motivierte, Informationen erweitert wurden, welche für die eigentliche Modelltransformation nicht relevant sind.

Sämtliche Referenzen auf andere Objekte werden standardmäßig auf BOTL-Objektassoziationen abgebildet. Lediglich Attributwerte vom Typ `String`, `StringBuffer` und die Java-Klassen zur Repräsentation primitiver Typen (z.B. `Integer`, `Boolean`, etc.) werden als primitive Werte behandelt. Alle Klassen, die das Java-Interface `java.util.Collection` direkt oder indirekt (durch Vererbung von Klassen und Interfaces) implementieren werden standardmäßig als Container-Klassen

für Mehrfachassoziationen interpretiert. Diese Annahmen sind für die Realisierung eines Prototypen vollkommen ausreichend, im Bedarfsfall kann die Zuordnung von Java-Klassen zu Mehrfachassoziationen jedoch ohne weiteres an individuelle Erfordernisse angepasst werden. Ein entsprechender Mechanismus zur Zuordnung von Java-Klassen zu BOTL-Typen ist bereits vorhanden.

6.5.3 Auffinden gültiger Matches

Um objektorientierte Modelle transformieren zu können, müssen zunächst Matches der Quellmodellvariablen in diesen Modellen gefunden werden. So muss für jede Regel die Menge $MFM(mf, mv)$ (siehe Definition 3.5.3, 84) aller Matches im Quellmodell ermittelt werden. Der hierzu benötigte Algorithmus ist in einer eigenen Klasse gekapselt, um ihn bei Bedarf durch eine, für ein gegebenes Anwendungsfeld effizientere, Implementierung ersetzen zu können. Das Interface

`de.tum.in.botl.transformer.interfaces.MatchfinderInterface`

muss von allen Klassen die für die Mustersuche eingesetzt werden sollen implementiert werden. Das Interface sieht lediglich zwei Methoden vor:

`Vector findMatches(Modelfragment mf, Modelvariable mv)` erhält als Aufrufparameter ein zu durchsuchendes Modellfragment und eine Modellvariable, deren Typ mit dem Metamodell des Modellfragments identisch sein muss. Als Ergebnis der Suche wird ein Vektor aus Objekten des Typs `Match` zurückgeliefert. Ein Match enthält zwei Tabellen, in denen jeder Objektvariablen bzw. Objektvariablenassoziation der Modellvariable `mv` genau ein Objekt bzw. eine Assoziation aus `mf` zugeordnet wird. Hierbei wird sichergestellt, dass das so gefundene Teilmodellfragment einen gültigen Match gemäß Definition 3.5.2 (siehe S. 83) darstellt.

`Vector getWarnings()` liefert einen Vector mit Warnhinweisen, die während der Suche erzeugt wurden. Die vorliegende Implementierung erzeugt einen Warnhinweis, falls ein Objektvariablenidentifikator der Quellmodellvariable ein Tupel ist und dieser bei der Suche nach Matches als $\diamondsuit$ interpretiert wurde.

Mustersuche in Objektgeflechten

Ein Match, wie er in Definition 3.5.2 (S. 83) eingeführt wurde, wird innerhalb des BOTL-Werkzeuges jeweils in einem Objekt der Klasse

`de.tum.in.botl.transformer.implementation.Match`

repräsentiert. Diese Klasse verfügt über vier Attribute, in denen alle zu einem Match gehörigen Informationen enthalten sind. Auf alle diese Attribute kann ausschließlich über geeignete Getter- und Setter-Methoden zugegriffen werden:

`Hashtable objMatches` In dieser Tabelle wird jeder Objektvariablen der entsprechenden Modellvariablen ein Objekt des Quellmodells zugeordnet.

`Hashtable assoMatches` Die Tabelle ordnet jeder Objektvariablenassoziation der Modellvariablen eine Objektassoziation des Quellmodells zu.

`Hashtable variableValues` Diese Tabelle enthält als Schlüssel sämtliche Variablennamen aus Attributvariablen und Identifikatortermen der Quellmodellvariablen. Die Werte sind die Werte, die sich für die Variablen für den jeweiligen Match im Quellmodell ergeben.

`boolean isValid` Dieser Boolsche Wert gibt an, ob der Match gültig ist.

Die vom Prototypen zur Mustersuche verwendete Klasse

```
de.tum.in.botl.transformer.implementation.Matchfinder
```

geht nach dem folgenden Schema vor:

1. Eine Liste von Matches mit allen möglichen eindeutigen Zuordnungen von Objektvariablen der Modellvariable auf Objekte des Modellfragments mit passendem Typ wird erstellt. Alle diese Matches sind als gültig gekennzeichnet und verfügen lediglich über Einträge in der Tabelle `objMatches`.
2. Für jeden dieser Matches wird untersucht, ob für jede Objektvariablenassoziation zwischen zwei Objektvariablen der Objektvariablen eine passende Objektassoziation zwischen den gematchten Objekten im Modellfragment existiert. Ist dies für eine Objektvariablenassoziation nicht der Fall, so wird der gesamte Match als ungültig markiert, d.h. der Wert des Attributs `isValid` wird auf `false` gesetzt.
3. Für jeden gültigen Match wird überprüft, ob er alle Constraints der Quellmodellvariablen erfüllt. Diese Überprüfung ist im nachfolgenden Abschnitt detaillierter beschrieben. Matches, die Constraints verletzen werden ebenfalls als ungültig markiert.
4. Die verbleibenden gültigen Matches werden in einem Ergebnisvektor zusammengefasst und als Rückgabewert übergeben.

Überprüfen von Constraints in Matches

Ein Match in einem Quellmodell ist nur dann gültig, falls er auch alle durch die Terme in Objektvariablenidentifikatoren und -attributen implizit festgelegten Constraints erfüllt.

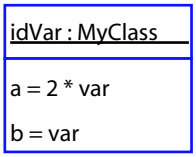


Abbildung 6.12: Eine Quellmodellvariable die einen Constraint formuliert

Beispielsweise wird durch die in Abbildung 6.12 dargestellte, nur aus einer Objektvariable bestehende, Quellmodellvariable der Constraint formuliert, dass ein Objekt nur dann ein gültigen Match darstellt, wenn sein Attribut *a* den doppelten Wert wie das Attribut *b* aufweist.

Die Methode

```
private void checkConstraints(Match match)
```

der Klasse `Matchfinder` überprüft für jeden gefundenen Match ob alle Constraints, die durch Terme in Identifiaktoren und Attributen spezifiziert wurden, erfüllt sind. Sind für einen Match nicht sämtliche Constraints erfüllt, so wird dieser als ungültig deklariert.

Zur Überprüfung von Constraints wird über alle Instanzen des Typs `TermOwner` aller Objektvariablen (also alle Objektvariablenidentifikatoren und Attributterme der Modellvariable) iteriert. Aus allen Attribute und Identifikatoren, deren Typ weder ein String noch ein Boolscher Typ ist, wird ein Gleichungssystem der Form

Gefundener Attribut-/Identifkatorwert = Term im Attribut/Identifikator

aufgebaut. Die Unbekannten des Systems sind hierbei sämtliche Variablen aus den Termen der Modellvariable. Die Klasse `EquationalSystemSolver` (siehe Abschnitt 6.5.1) wird für die Lösung dieses Gleichungssystems verwendet. Ist dieses Gleichungssystem eindeutig lösbar, so sind alle Constraints offensichtlich erfüllt, andernfalls wird der Match als ungültig gekennzeichnet.

Attribute vom Typ `String` können lediglich kopiert werden. Dementsprechend dürfen Attribute dieses Typs in Quellobjektvariablen entweder einen konstanten `String`-Wert oder aber eine einzelne Variable enthalten. Der Grund hierfür ist, dass Zeichenketten keine mathematische Struktur bilden die für das Lösen von Gleichungssystemen geeignet ist. Die Korrektheit der gefundenen `String`-Werte wird dementsprechend gesondert behandelt und geht nicht mit in das Gleichungssystem ein. Auch für Attribute von einem Boolschen Typ erfolgt eine entsprechende Sonderbehandlung.

Der Wert jeder Variablen wird schließlich dem Variablennamen als Schlüssel in die Tabelle `variableValues` des `Match`-Objektes eingetragen. Diese Werte werden später für die Berechnung von Attribut- und Identifikatorwerten in Zielmodellfragmenten während einer Modellfragmenttransformation herangezogen.

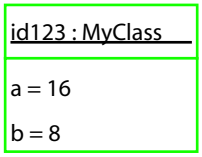


Abbildung 6.13: Ein Modellfragment das zusammen mit der Quellmodellvariable aus Abbildung 6.12 einen Match bildet.

Ein Match der in Abbildung 6.12 dargestellten Modellvariable könnte das in Abbildung 6.13 dargestellte Modellfragment beinhalten. Dieser Match würde demnach zu dem folgendem Gleichungssystem führen:

$$\begin{aligned} \text{id123} &= \mathit{idVar} \\ 16 &= 2 \cdot \mathit{var} \\ 8 &= \mathit{var} \end{aligned}$$

Das System hat offensichtlich die eindeutige Lösung:

$$\begin{aligned} \mathit{idVar} &= \text{id123} \\ \mathit{var} &= 8 \end{aligned}$$

Demnach handelt es sich hierbei um einen gültigen Match.

6.5.4 Transformation von Modellfragmenten

Implementierungen des Interfaces `MatchfinderInterface` liefern einen Vektor von Match-Objekten, die jeweils einen Match der jeweiligen Quellmodellvariablen einer Regel in ihrem Quellmodell enthalten. Aus jedem dieser gefundenen Fragmente wird nun entsprechend der Regel ein neues Zielmodellfragment erzeugt. Hierzu steht die Methode

```
private ModelFragment mft(Match match, Rule r)
```

zur Verfügung, welche die in Definition 3.5.5 (S. 89) eingeführte Abbildung $mft(mfm_0, r)$ implementiert. Die Methode erhält als Parameter einen Match einer linken Regelseite in einem Modell und die zugehörige Regel. Als Ergebnis liefert `mft()` das durch die Regel erzeugte Zielmodellfragment.

Bei der Erzeugung eines neuen Zielmodellfragments wird zuerst ein neues Modellfragment mit derselben Struktur wie die der Zielmodellvariablen der Regel `r` erzeugt. Im Anschluss werden zunächst für alle Attribute der Objekte im neuen Zielmodellfragment Werte berechnet.

Die Terme von Objektvariablenidentifikatoren werden mittels der Methode

```
private String evaluateIDTerm(Term term, Match match)
```

ausgewertet. Die Methode erhält als Parameter den auszuwertenden Term vom Typ `Term` und den Match auf der linken Regelseite. Konstante Werte können hierbei direkt angegeben werden, für den Term $\diamondsuit$ (im Werkzeug durch das Zeichen ♯ repräsentiert) wird ein eindeutiger Identifikator generiert.

Zur Berechnung des Wertes von Termen des Typs `Variable` und `Expression` wird nach dem folgenden Muster vorgegangen:

- Der Term wird geparst, um eine Symboltabelle aller im Term enthaltenen Variablen zu erhalten. Hierzu wird die unter GNU-Lizenz verfügbare Java-Bibliothek JEP (Java Expression Parser) [Fun04] verwendet. JEP ist eine Java-API zum Parsen und Auswerten von mathematischen Ausdrücken.

- Für jede gefundene Variable wird aus der Tabelle `variableValues` des `Match`-Objektes der entsprechende Wert der Variablen gelesen und dem JEP-Parser als Variablenwert übergeben. Da JEP auch `String`-Werte verarbeiten kann, werden auch Zeichenketten als `String`-Objekte an den Parser übergeben. Dies ermöglicht es beispielsweise auch Boolsche Werte durch den Vergleich von Zeichenketten zu berechnen. Alle anderen Werte werden in das Java-Format `double` konvertiert, das von JEP intern für die Auswertung von Ausdrücken verwendet wird.

- Der JEP-Parser wertet den Term aus und das Ergebnis wird als Rückgabewert übergeben.

Für die Auswertung von Identifikatortermen vom Typ `Tupel` wird für jedes Tupelelement die Methode `evaluateIDTerm()` rekursiv aufgerufen und das zusammengesetzte Ergebnis zurückgegeben.

Die Werte die sich für Attributvariablen werden durch die Methode

```
private String evaluateAttTerm(Term term, Match match)
```

berechnet. Im Gegensatz zu Identifikatoren wird für den Wert $\diamondsuit$ hier der Wert `null` zurückgeliefert. Da Tupel nicht als Terme von Objektvariablenattributen erlaubt sind, brauchen diese hier nicht weiter berücksichtigt zu werden. Ansonsten geschieht die Auswertung der Attributterme analog zu der von Identifikatortermen.

Falls die Auswertung eines Terms fehlschlägt, wird eine Ausnahme des Typs `TransformationException` mit der jeweiligen Fehlerinformation geworfen. Dies ist insbesonders dann der Fall, wenn es nicht möglich ist, einer im auzuwertenden Term vorkommenden Variablen einen Wert in der Tabelle `variableValues` des Matches zuzuordnen. In diesem Fall existiert auf der rechten Regelseite eine Variable, die nicht auf der linken Regelseite vorkommt, die Transformation schlägt dementsprechend fehl, da das betreffende Regelwerk nicht anwendbar (siehe Abschnitt 4.1, S. 110ff.) ist.

6.5.5 Einfügen neuer Fragmente in das Zielmodell

Jedes neu erzeugte Zielmodellfragment wird nach seiner Erzeugung in das bestehende Zielmodell eingefügt. Im Gegensatz zum $\cup_m$-Operator aus Definition 3.5.12 (siehe S. 95) hat die entsprechende Methode

```
private void mergeIntoTarget(ModelFragment mf)
```

jedoch nur einen Parameter, da als zweiter Parameter immer das sich im Aufbau befindliche Zielmodellfragment angenommen wird. Hierdurch wird das speicher- und rechenintensive Erzeugen immer neuer Modellfragmente vermieden.

Im Verlauf des Einfügens des neuen Modellfragments wird die Methode

```
private void mergeObjects(BotlObject o0, BotlObject o1)
```

verwendet, um den Inhalt des neu erzeugten Objektes `o1` in ein ggf. bestehendes Objekt `o0` einzufügen. Die Methode verhält sich hierbei so, wie es in Definition 3.5.11 von *OBmerge* (S. 94) gefordert ist. Dieser Vorgang kann bei nicht anwendbaren Regelwerken fehlschlagen, wenn versucht wird einen bestehenden Attributwert mit einem anderen Wert zu überschreiben. In diesem Fall wird eine Ausnahme des Typs `MergeException` mit ausführlichen Fehlerinformationen geworfen.

6.5.6 Export von Java-Objektgeflechten

Nach Abarbeitung aller Regeln steht das erzeugte Zielmodell in Form eines BOTL-Modells zur Verfügung. Um in der Praxis nutzbar zu sein, muss dieses Modell wieder in ein technologiespezifisches Format umgewandelt werden. Analog zum Importer existiert hierfür das Interface

`de.tum.in.botl.adapter.Exporter`,

welches zwei Methoden für einen BOTL-Exporter festlegt:

`Enumeration start(ModelFragment mf)` exportiert das übergebene Modellfragment in ein externes Format und liefert eine Enumeration die (in Abhängigkeit von der jeweiligen Zielplattform) Verweise auf sämtliche exportierte Objekte enthält.

`Object start(ModelFragment mf, BotlObject bObj)` exportiert ebenfalls das übergebene Modellfragment, liefert jedoch nur einen Verweis auf das Objekt, das aus dem BOTL-Objekt `bObj` erzeugt wurde. Dieses kann, bei geeigneter Wahl, als Ausgangspunkt für die Navigation durch das erzeugte Modellfragment dienen.

Für den Prototypen wurde eine Implementierung zum Export von BOTL-Modellen nach Java erstellt. Die Klasse

`de.tum.in.botl.adpater.implemention.JavaExporter`

erzeugt für jedes Objekt eines BOTL-Modellfragments ein entsprechendes Java-Objekt. Im Anschluss werden für alle Objektassoziationen im BOTL-Modell Referenzen zwischen den erzeugten Java-Objekten erzeugt. Wie beim Import von Java-Objektgeflechten werden auch hier alle Klassen, die direkt oder indirekt das Interface `java.util.Collection` implementieren, als Container-Klassen für Mehrfachassoziationen interpretiert. Die Identitäten der Java-Objekte werden in einer Hash-Tabelle auf die der BOTL-Objekte abgebildet, um die Enden von Assoziationen im Objektgeflecht eindeutig identifizieren zu können.

6.5.7 Verwendung des Transformators

Das folgende Code-Beispiel zeigt wie sich die Transformationskomponente in eine bestehende Anwendung einbinden lässt. Im Beispiel wird davon ausgegangen, dass innerhalb des XML-Dokuments „MyRuleSet.botl.xml“ eine Modelltransformation mit einem einzigen Quellmetamodell mit dem Namen „TheSourceMetamodel“ spezifiziert ist. Es soll nun ein Java-Objektgeflecht transformiert werden, dessen Objekte alle von dem Objekt `srcRootObj` aus erreichbar sind. Der Code für die Ausführung dieser Transformation hat demnach die folgende Gestalt:

```
Transformer trafo    = new Transformer("MyRuleSet.botl.xml");
Importer    importer = new JavaImporter("TheSourceMetamodel");
Exporter    exporter = new JavaExporter();

Enumeration result = exporter.start( trafo.transform(
                                importer.start(srcRootObj)) );
```

Die zurückgelieferte `Enumeration` enthält alle erzeugten Objekte. Selbstverständlich sind auch alle Referenzen der erzeugten Objekte untereinander gesetzt.

Die Klasse

```
de.tum.in.botl.transformer.JavaTransformer
```

enthält zudem eine generische Implementierung eines Transformators für Java-Objektstrukturen. Diese erlaubt es, durch den Aufruf der statischen Methode

```
public static Enumeration transform( String botlFile,
                                     Hashtable hooks )
```

Java-Objektgeflechte zu transformieren. Der Parameter `botlFile` enthält hierbei den Pfad zu dem XML-Dokument, in dem die Transformation spezifiziert ist. Die Tabelle `hooks` enthält als Schlüssel jeweils den Namen eines Quellmetamodells und als Wert eine Referenz auf ein Java-Objekt, über das ein Java-Modell mit diesem Metamodell erreichbar ist.

6.6 Verwandte Werkzeuge und Ansätze

Im Umfeld von Graphgrammatiken ist die Transformation von Modellen schon seit längerem ein vieldiskutiertes Thema. Doch gerade mit der zunehmenden Popularität der Model Driven Architecture gewann dieses Thema in jüngster Zeit verstärkt an Bedeutung. Dementsprechend existiert mittlerweile eine Vielzahl von Ansätzen und prototypischen Werkzeugen zur Modelltransformation. Im Rahmen dieses Abschnitts werden einige dieser Ansätze, für die eine mit dem BOTL-Werkzeug vergleichbare Werkzeugunterstützung existiert, kurz vorgestellt und disskutiert.

6.6.1 PROGRES

PROGRES (PROgrammierte GRaph-Ersetzungssysteme) [Sch91] ist eine von Prof. Dr. Andy Schürr an der RWTH Aachen entwickelte Spezifikationssprache für Graphersetzungssysteme. Für die Sprache existiert eine Werkzeugunterstützung für Linux- und Solaris-Systeme. PROGRES ist das derzeit bekannteste und umfangreichste Graphersetzungssystem und wurde bereits in einer Reihe wissenschaftlicher Studien eingesetzt [Lef94, LS96, RS97, Sch88].

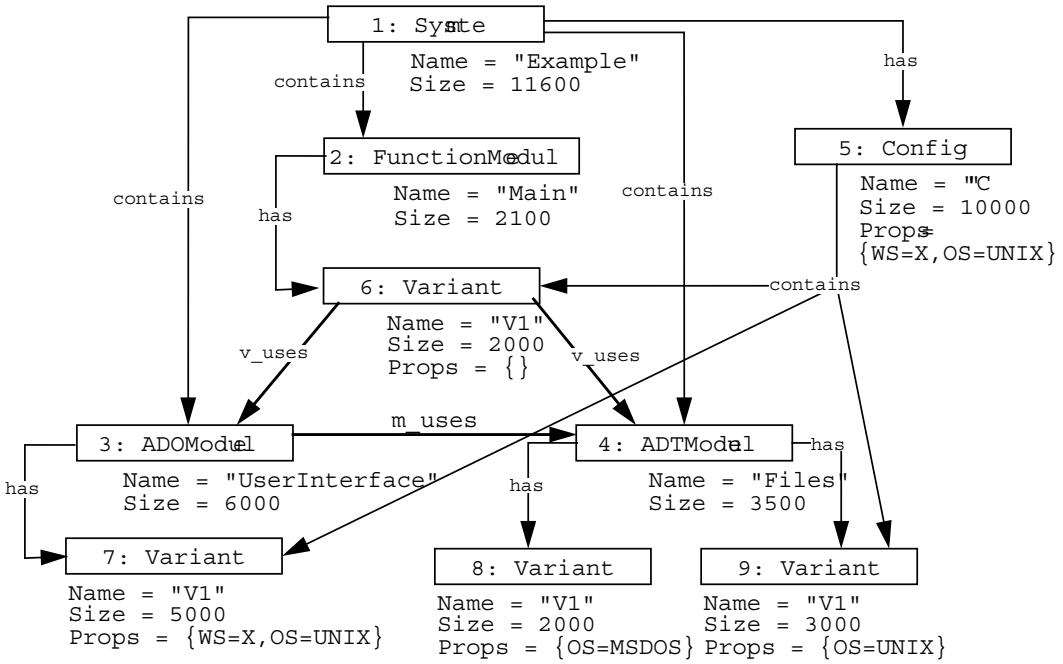


Abbildung 6.14: Beispiel für die interne Darstellung eines Graphen in PROGRES [Sch97]

Graphmodelle und Graph-Schemas

Bei dem Graphmodellen die als PROGRES-Datenstrukturen dienen handelt es sich um gerichtete, attributierte Graphen mit beschrifteten Knoten und Kanten. Ein Graph besteht demnach aus:

Knoten Ein Knoten kann über eine Menge von Attributen verfügen und ist mit einem Typ beschriftet.

Attribute Ein Attribut verfügt über einen Namen und einen Wert. Handelt es sich bei einem Attribut um ein abgeleitetes Attribut (*derived attribute*), so errechnet sich dessen Wert automatisch aus einer Berechnungsvorschrift für den jeweiligen Typ des Attributs.

Kanten Kanten verbinden Knoten. Sie sind gerichtet und mit dem Namen ihres Typs beschriftet. Im Gegensatz zu Knoten verfügen Kanten über keine Attribute.

Abbildung 6.14 zeigt die interne Darstellung eines PROGRES-Graphen. Das Beispiel ist [Sch97] entnommen.

Eine Menge gültiger Graphen wird in PROGRES durch ein Graph-Schema definiert. Hierfür sieht PROGRES sowohl eine graphische, als auch eine mächtigere textuelle Notation vor. Die wesentlichen Elemente von Graph-Schemata sind im einzelnen:

Knotenklassen Durch Knotenklassen werden Mengen von Knotentypen definiert. Zwischen Knotenklassen können Vererbungsbeziehungen bestehen, auch Mehrfachvererbung wird unterstützt. Für Knotenklassen können Attribute definiert werden. Für geerbte, abgeleitete Attribute kann für jede Knotenklasse eine eigene Berechnungsvorschrift angegeben werden.

Attributtypen Innerhalb von PROGRES existiert ein Reihe vordefinierter primitiver Typen für Attribute wie `integer` oder `boolean` zusammen mit einer Menge von Operationen auf diesen Typen. Weitere Typen können unter Verwendung einer Programmiersprache spezifiziert

werden. Für die Spezifikation zusätzlicher Operationen über den primitiven Datentypen bietet PROGRES eine eigene, textbasierte Sprache an.

Kantentypen Kantentypen sind gerichtet und verbinden Knotenklassen. Ein Kantentyp legt fest, dass in einem gültigen Graphen eine Kante des jeweiligen Typs zwischen Knoten des passenden Typs bestehen darf. Für jedes Ende eines Kantentyps kann die mögliche Kardinalität der ein- bzw. ausgehenden Kanten durch die Angabe einer Multiplizität angegeben werden. Erlaubt sind hierfür jedoch lediglich die vier Ausdrücke [0, 1], [0, *], [1, 1] und [1, *].

Knotentypen Ein Knotentyp ist jeweils eine Instanz einer Knotenklasse. Die Instanzen von Knotentypen sind Knoten.

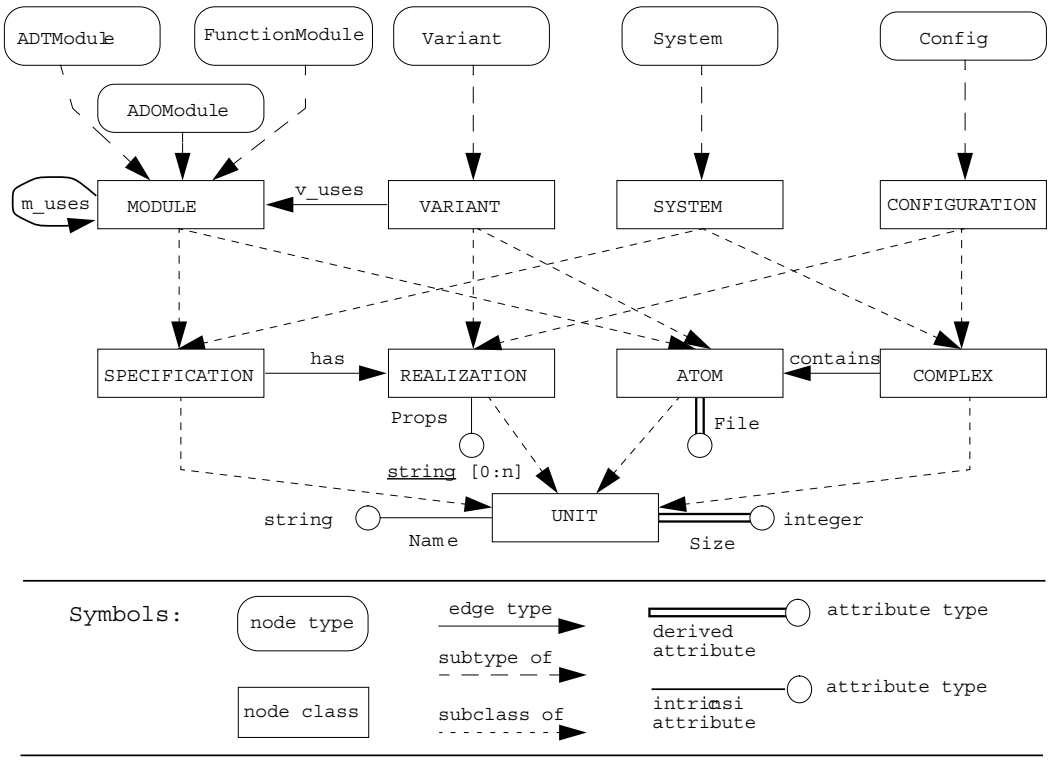


Abbildung 6.15: Beispiel für ein zum Graphen aus Abbildung 6.15 passendes PROGRES-Schema [Sch97]

Abbildung 6.15 stellt ein, zu dem Graphen aus Abbildung 6.14 passendes, PROGRES-Graph-Schema dar. Die Knotentypen des Graphen sind in der Abbildung oben als Kästen mit runden Ecken dargestellt, Graphklassen entsprechen eckigen Kästen. Die Bedeutung der anderen graphischen Elemente kann der Legende auf der Unterseite der Abbildung entnommen werden.

Graph-Schemata stellen somit Metamodelle für Graphen dar. Die hier verwendeten Konzepte wie Vererbung und die Möglichkeit Multiplizitätsgrenzen für Kanten (wenn auch nur in eingeschränkter Form) anzugeben sind praktisch mit denen von in BOTL verwendeten Klassenmodellen identisch.

Auffinden von Subgraphen

Das Auffinden von Subgraphen in einem Ursprungsgraphen, wie es in PROGRES möglich ist, entspricht der Suche nach Matches in einem Quellmodell innerhalb von BOTL.

PROGRES stellt eine graphische Spezifikationssprache für die Überprüfung von Eigenschaften von Subgraphen (sog. *Tests*) zur Verfügung. Mit Hilfe dieser Notation können Muster für Subgraphen, nach denen ein Graph durchsucht werden soll, angegeben werden. Ein Beispiel für eine solche Spezifikation ist in Abbildung 6.16 angegeben.

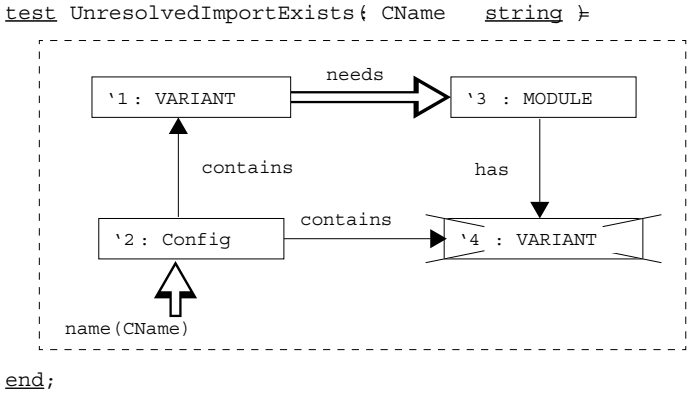


Abbildung 6.16: Beispiel für einen PROGRES-Test [Sch97]

Der Knoten vom Typ VARIANT im Beispiel ist ein negativer Kontext. D.h. für einen gefundenen Subgraph darf der Graph an dieser Stelle *keinen* Knoten dieses Typs enthalten. Der mit needs beschriftete Doppelpfeil symbolisiert einen Constraint zwischen den beiden Knoten die er verbindet. Er besagt, dass das die Relation needs für die beiden gefundenen Knoten gelten muss. Eine solche Relation wird innerhalb von PROGRES *Pfad* genannt und wird textuell spezifiziert. Mit Hilfe dieser Konstrukte ist es beispielsweise auch möglich Aussagen über die transitive Hülle einer Graphstruktur zu machen oder das letzte Element einer Kette zu identifizieren. Gleiche Pfeile an nur einem Knoten bezeichnen Prädikate für den jeweiligen Knoten (*Restrictions*), die ebenfalls mit Hilfe einer PROGRES-eigenen Sprache definiert werden.

Neben der Möglichkeit PROGRES-Tests von Subgraphen graphisch zu spezifizieren existiert auch eine textuelle, imperative Sprache zur Formulierung dieser Anfragen.

Auf Basis der Test-Spezifikationen lassen sich mit Hilfe einer Prolog-ähnlichen Sprache komplexe Abfragen nach Teilgraphen (*Queries*) formulieren. Somit erlaubt PROGRES das Auffinden von Subgraphen innerhalb eines Graphen anhand einer sehr ausdrucksmächtigen Abfragesprache.

Transformation von Graphen

Die Transformation eines bestehenden Graphen erfolgt durch die Ausführung einer Reihe von *Produktionen*. Die Spezifikation einer Produktion setzt sich im wesentlichen aus einer linken und einer rechten „Regel“-Seite zusammen. Jede Seite einer Produktion enthält ein Graphmuster, wie sie auch von Tests zum Auffinden von Graphen verwendet werden. Abbildung 6.17 stellt exemplarisch die Spezifikation einer Produktion dar.

Neben den klassischen Musterknoten, die auf genau einen Knoten des Graphen matchen müssen, können die linken Seiten von Produktionen auch spezielle Arten von Musterknoten enthalten. Diese müssen je nach Art des Musterknotens jeweils auf einen oder keinen, eine ggf. leere oder aber eine nicht-leere Menge von Knoten mit den geforderten Eigenschaften matchen.

Produktionen sind parametrisierbar, d.h. sie können Paramter erhalten in denen beispielsweise der Typ eines Musterknotens angegeben wird. Dies erlaubt es, sehr allgemeine Produktionen anzugeben, die in verschiedenen Kontexten genutzt werden können.

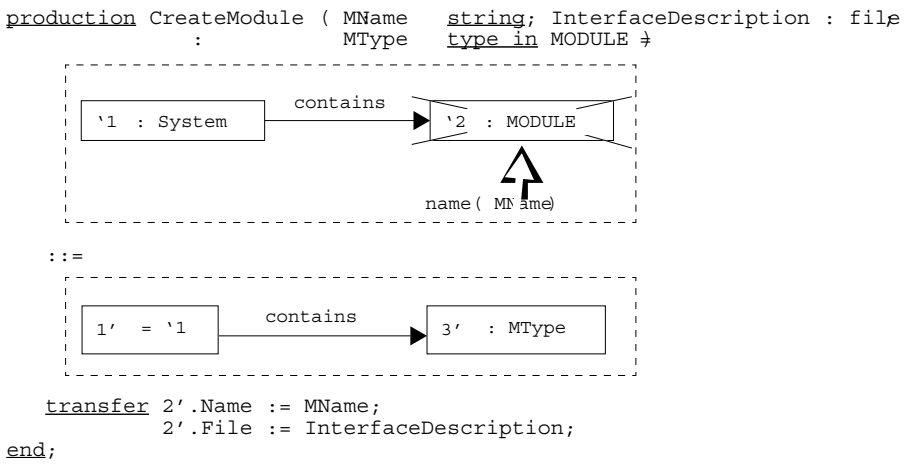


Abbildung 6.17: Beispiel für eine PROGRES-Produktion [Sch97]

Tests und Produktionen sind die Bausteine, aus denen komplexe Transformationen (*Transactions*) gebildet werden können. Hierbei kommt dieselbe imperative Sprache wie sie auch für die Spezifikation von Tests verwendet wird zum Einsatz. Diese enthält neben Konstrukten zur Ablaufsteuerung auch nicht-deterministische Sprachelemente, wie z.B. die nicht-deterministische Verzweigung in der Programmausführung.

Transactions sind atomar, sie gelingen entweder als Ganzes oder lassen den Graphen unverändert. Hierdurch wird sichergestellt, dass ausschließlich gültige Graphen bezüglich des verwendeten Graph-Schemas erzeugt werden. Durch Backtracking wird weiterhin sichergestellt, dass eine Transaction nie fehl schlägt, falls eine fehlerfreie Durchführung möglich ist.

Zusammenfassung

PROGRES ist eine mittlerweile langjährig bewährte Sprache zur Graphtransformation deren Realisierbarkeit anhand des PROGRES-Werkzeugs nachgewiesen wurde. Im Gegensatz zu BOTL führt PROGRES Rewrite-Transformationen auf einem bestehenden Graphen aus, d.h. ein Graph wird durch die Anwendung einer Reihe von Produktionen schrittweise verändert. Die durch das Werkzeug unterstützte Sprache zeichnet sich durch eine sehr große Ausdrucksmächtigkeit aus. So ist es, im Gegensatz zu BOTL, mit PROGRES möglich negative Kontexte für die Auswahl von Teilen des Quellgraphen anzugeben oder aber komplexe Suchmuster, wie z.B. die transitive Hülle, zu formulieren.

Hierzu bietet PROGRES eine Reihe von graphischen und textuellen Spezifikationssprachen, die jeweils unterschiedliche Paradigmen, wie imperative, relationale oder regelbasierte Ansätze, verwenden.

Diese Fülle an Ausdrucksmöglichkeiten ist jedoch auch mit einer Reihe von Nachteilen verbunden. In erster Linie ist hier die hohe Komplexität der Sprache anzuführen. Das Erstellen von PROGRES-Spezifikationen erfordert zunächst einen sehr hohen Einarbeitungsaufwand. Die Kombination unterschiedlichster Sprachkonzepte lässt die Sprache insgesamt nur wenig intuitiv und schwer beherrschbar erscheinen.

Die verwendeten Graphmodelle und Graph-Schemata sind prinzipiell sehr eng mit den Konzepten der Objektorientierung verwandt. Jedoch sind die verwendeten graphischen Beschreibungstechniken für sie sehr unüblich und für Entwickler die objektorientierte Beschreibungstechniken wie die UML eher verwirrend.

Im Gegensatz zu BOTL ist es für eine PROGRES-Spezifikation nicht möglich nachzuweisen, dass sie für eine Klasse von Eingabegraphen immer anwendbar ist. Dies entscheidet sich hier erst zur Laufzeit, wobei im Fehlerfall eine Transformation als ganzes fehlschlägt und keine Auswirkungen auf den Graphen hat.

Dieser Umstand und die hohe Komplexität der Sprache sind wohl die Hauptgründe dafür, dass PROGRES zwar ein dominierende Stellung bei den Ansätzen zur Graphtransformation einnimmt, sich jedoch in der Praxis nie auf breiter Front etablieren konnte.

6.6.2 OPTIMIX und GREAT

OPTIMIX [Aßm98b, Aßm98a, Aßm99] ist ein am Institut für Programmstrukturen und Datenorganisation von Prof. Dr. Goos an der Universität Karlsruhe entwickeltes Werkzeug zur Generierung von Code für die Analyse und Transformation von Programmen. Das ursprüngliche Einsatzfeld von OPTIMIX ist die Optimierung von Zwischencode, wie er von Compilern erzeugt wird. Hierbei werden die Zwischencodestrukturen als Graphen interpretiert, die auf Basis von OPTIMIX-Regeln analysiert und transformiert werden.

OPTIMIX unterstützt zwei Arten von Graph-Rewrite-Mechanismen, Edge Addition Rewrite Systems (EARS) [Aßm94] und Exhaustive Graph Rewrite Systems (XGRS) [Aßm96]. In der neusten verfügbaren Version 2.5 von 1998 wird auch die Transformation von Java-Code unterstützt, was OPTIMIX auch als mögliche Ausführungsumgebung für BOTL-Regelwerke interessant erscheinen lässt. Datenstrukturen werden hierbei direkt als Java-Klassen spezifiziert. Zusätzlich werden Regeln in einer textuellen Syntax angegeben aus der das Werkzeug Java-Code zur Transformation der Objektstrukturen erzeugt.

So wäre es möglich BOTL-Regeln in OPTMIX-Spezifikationen umzuwandeln und aus diesen den Java-Code zur Transformation von Java-Objektgeflechten oder zumindest für das Pattern-Matching in den Ursprungsmodellen zu generieren. Allerdings hat sich gezeigt, dass eine Verwendung von OPTIMIX nicht möglich ist, da das Werkzeug keinen lauffähigen Java-Code erzeugt und die Entwicklung mittlerweile nicht mehr weiter betrieben wird. Auch die Beschränkung auf SUN OS4 und Linux als Plattform sprechen gegen eine Verwendung von OPTIMIX zur Code-Erzeugung.

Derzeit existiert ein Vorhaben am Forschungszentrum Informatik in Karlsruhe OPTIMIX als Grundlage für die Entwicklung des Modelltransformationswerkzeugs GREAT[1] zur Transformation von UML-Modellen einzusetzen [Chr02]. Modelltransformationen werden in GREAT ebenfalls ausschließlich textuell spezifiziert. Auch eine Überprüfung, ob eine Transformationsspezifikation ausschließlich metamodellkonforme Zielmodelle erzeugt ist nicht vorgesehen. Zudem ist das Vorhaben noch in einem vergleichsweise frühen Stadium, so dass derzeit keine lauffähige Implementierung des Werkzeuges verfügbar ist.

6.6.3 GReAT

GReAT (Graph-REwriting And Transformation) [AKSS03] ist eine, gemäß [AKS03], auf der UML basierende Sprache zur Graphtransformation. Die Sprache und eine zugehörige Werkzeugunterstützung [Agr04] wurden an der Vanderbilt University Nashville entwickelt. Ziel von GReAt ist es, einen Mechanismus zur Transformation von Objektmodellen, deren Struktur anhand von UML-Klassendiagrammen festgelegt ist, zu schaffen.

Demzufolge entsprechen die in GReAT verwendeten Graphen im wesentliche Objektmodellen, für die jeweils Metamodelle in Form von Klassenmodellen vorliegen. GReAT erzeugt im Verlauf einer

[1] nicht zu verwechseln mit dem GReAT-Werkzeug aus Abschnitt 6.6.3

Transformation kein neues Modell, sondern transformiert wie die meisten an Graphgrammatiken orientierten Ansätze ein bestehendes Modell. Daher existiert für die Transformation ein gemeinsames Metamodell, welches sich aus dem Ziel- und dem Quellmetamodell, sowie ggf. zusätzlichen, für die Transformation benötigten Zwischenstrukturen, zusammensetzt.

Für die Durchführung einer Produktion erlaubt es GReAT, Muster (*Patterns*) aus Platzhaltern für Objekte und Assoziationen zu definieren. Um redundante Strukturen einfach modellieren zu können, bietet GReAT einen Makromechanismus, mit dem sich komplexe Patterns in einer verkürzten Schreibweise darstellen lassen.

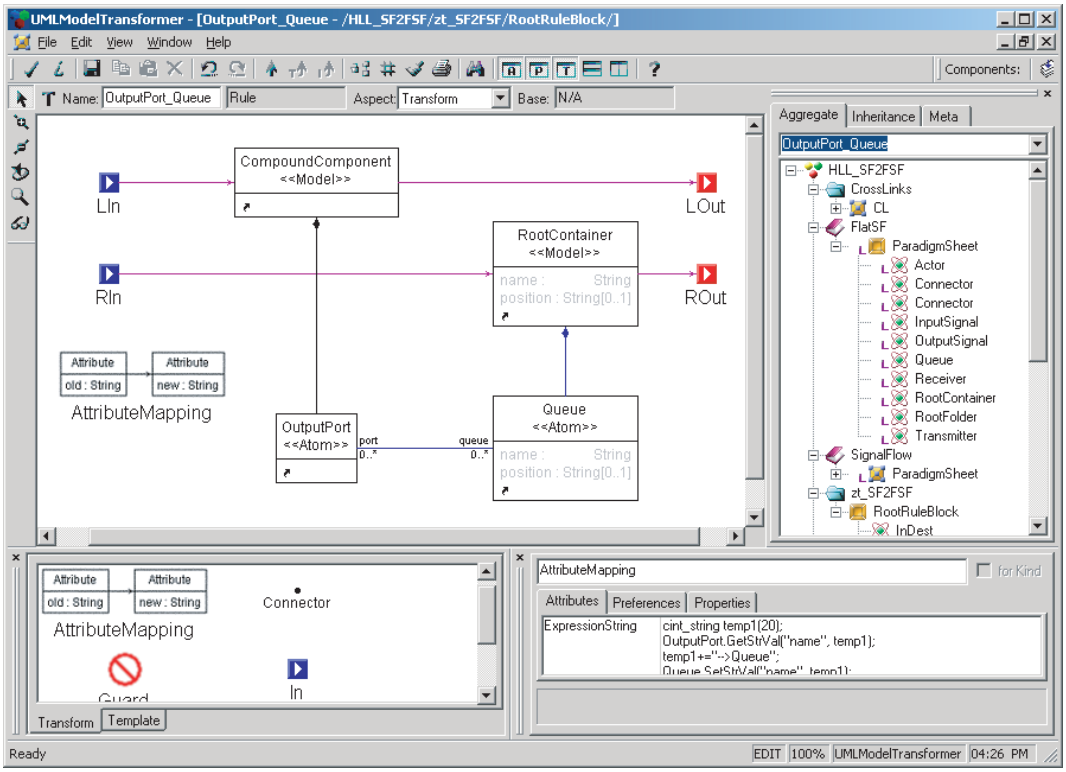


Abbildung 6.18: Eine Produktionsregel im GReAT-Werkzeug

Die Spezifikation einer Transformation erfolgt anhand einer Reihe von Produktionsregeln (*Productions*). Abbildung 6.18 zeigt das GReAT-Werkzeug, in dessen linken oberen Teil die Spezifikation einer Produktion dargestellt ist. Eine Produktion kann sich im wesentlichen aus den folgenden Elementen zusammensetzen:

- einem *Pattern*;
- einer *Guard- Condition*, die erfüllt sein muss damit die Regel ausgeführt werden kann;
- einer Menge von *Attribute Mappings*, d.h. Zuweisungsvorschriften die angeben wie sich neue Attributwerte errechnen und
- einer *Pattern Role*, einer Abbildung die jedem Knoten und jeder Kante des Patterns einen Wert der Menge $\{bind, delete, new\}$ zuordnet;

Während der Ausführung einer Produktion werden gemäß dem jeweiligen Pattern Teilgraphen im Graphen gesucht und ersetzt. Hierbei gilt jeweils:

- Mit *bind* markierte Elemente des Pattern müssen im gefunden Teilgraphen vorhanden sein und werden von der Produktion nicht verändert.

- Mit *delete* markierte Patternelemente müssen ebenfalls vorhanden sein und werden durch die Produktion aus dem Graphen entfernt.

- Mit *new* markierte Elemente müssen nicht vorhanden sein, sondern werden neu erzeugt.

Weiterhin verfügen Produktionsregeln, ähnlich wie Funktionen in Programmiersprachen, über Ein- und Ausgabeschnittstellen die mit Knoten ihres Patterns verbunden sind. Über die Eingabeschnittstelle können einzelnen Knoten eines Patterns explizit Objekte des jeweils zu transformierenden Objektmodells zugewiesen werden. Hierdurch verringert sich die Zahl der gefundenen Treffer eines Patterns in einem Objektmodell drastisch. Die Knoten des Patterns die Teil der Ausgabeschnittstelle sind können als Eingaben an eine nachfolgende Produktionsregel übergeben werden.

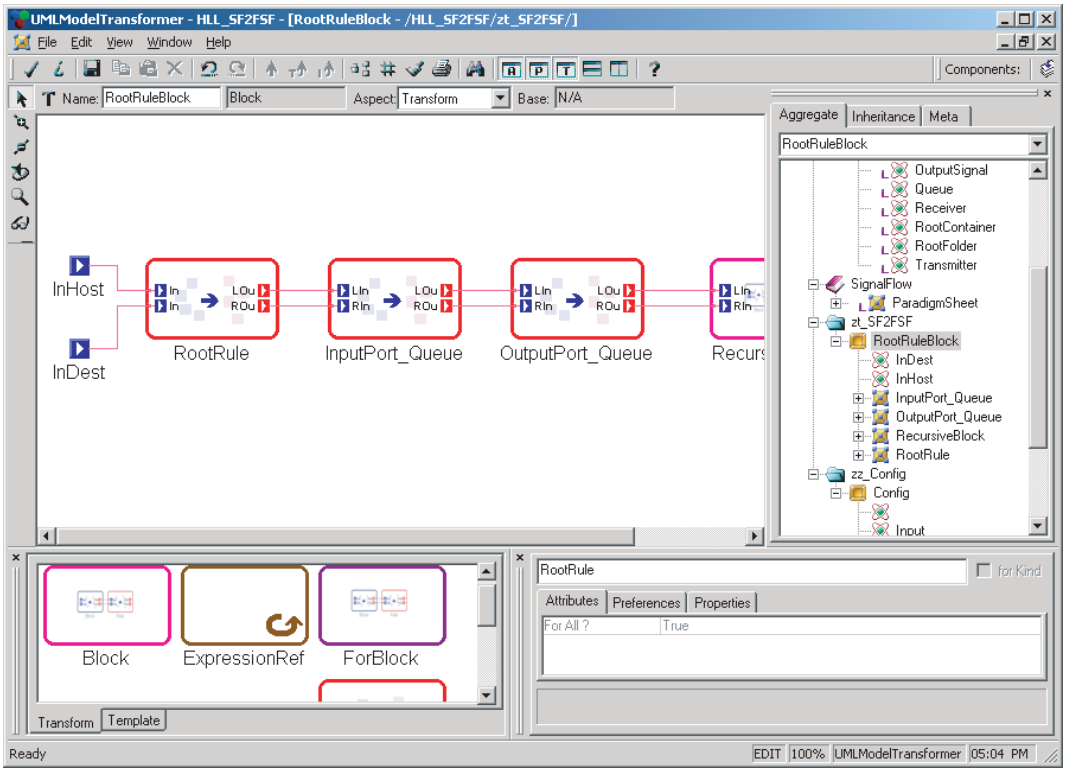


Abbildung 6.19: Sequentielle Ausführung von Produktionen

Abbildung 6.19 stellt die sequentielle Verkettung von Produktionen innerhalb des GReAT-Werkzeugs dar. Neben der sequentiellen Verkettung existieren hierarchisch organisierte Regeln sowie konditionelle und nicht-deterministische Verzweigungsoperatoren.

Grundsätzlich sind GReAT-Patterns lediglich eine verkürzte Schreibweise klassischer Graphgrammatiken. Die Ausdrucksmächtigkeit der Sprache wird jedoch durch zusätzliche Sprachkonzepte zur Ablaufsteuerung und Berechnung von Attributwerten erweitert. Diese können jedoch zugleich als Nachteil der Sprache angesehen werden kann. So ist die Anwendung einer einzelnen Produktion zwar noch intuitiv erfassbar, das Ergebnis der Ausführung einer GReAT-Spezifikation mit der Übergabe

von Ein- und Ausgabekontexten und einem komplexen Kontext ist jedoch zumeist nicht mehr einfach vorherzusagen. Dies ist umso schwerwiegender, da GReAT keine Mechanismen kennt um die Metamodellkonformität einer Spezifikation zu verifizieren.

Somit die führt Verwendung unterschiedlicher Sprachkonzepte für die Formulierung von Produktionen, Constraints, Kontrollfluss und Attributauswertungen in GReAT, ähnlich wie bei PROGRES, zu schwer lesbaren Spezifikationen, deren intuitive Interpretation in aller Regel Experten vorbehalten bleibt.

6.6.4 Generative Model Transformer (GMT) und UMLX

Im Umfeld der MDA ist derzeit das Open-Source-Werkzeugpaket GMT (Generative Model Transformer) in der Entstehung begriffen. Das Werkzeug ist von besonderem Interesse, da die Entwicklung von GMT ein Teil des von IBM inizierten Open- Source-Projektes Eclipse [Ecl04] ist, welches bereits eine weite Verbreitung gefunden hat. Das Ziel von Eclipse ist die Schaffung einer frei verfügbaren Software-Entwicklungsplattform. GMT soll als Teil von Eclipse einen Software-Entwicklungsprozess gemäß des MDA-Ansatzes unterstützten.

Zum Zeitpunkt der Erstellung dieser Arbeit existiert noch keine Implementierung oder Architekturbeschreibung des GMT-Werkzeuges. Lediglich eine erste Version der Anforderungsspezifikation für GMT ist derzeit verfügbar [Bet04]. Demzufolge soll GMT einen an die FAST-Methodik [WL99] angelehnten Entwicklungsprozess unterstützen, welcher die Software-Entwicklung von der Domänenanalyse bis hin zur Implementierung und dem Änderungsmanagement umfasst. Die geplante Realisierung von GMT gliedert sich dementsprechend in vier Hauptkomponenten zur Unterstützung dieses Prozesses:

- Eine Komponente zum Zusammenführen von zwei verschiedenen XMI-Modellen.
- Eine Komponente zur Transformation von XMI-Modellen.
- Eine Komponente zur Generierung von Anwendungs-Code aus XMI-Modellen.
- Eine Workflow-Komponente, welche diese Komponenten, das Eclipse-Framework und ggf. weitere Werkzeuge von Drittanbietern integriert.

Im Rahmen dieser Arbeit ist die Komponente zur Transformation von XMI-Modellen, welche letztendlich UML-Modelle repräsentieren, von besonderem Interesse. Da sich die Dokumentation zur Transformationssprache UMLX [Wil03b] auf der Projekt-Homepage findet, ist davon auszugehen, dass diese Sprache als Grundlage für die Realisierung der Fähigkeiten zur Transformation von Modellen Verwendung finden soll.

UMLX ist eine graphische Sprache zur Spezifikation von Transformationen von UML-Objektmodellen. Für die Spezifikation von Metamodellen wird dementsprechend eine eingeschränkte Variante von UML-Klassenmodellen verwendet. Im Gegensatz zu den auf Graphgrammatiken basierenden Sprachen und Werkzeugen, führt UMLX Model-to-Model-Transformationen aus, d.h. ein bestehendes Quellmodell wird nicht verändert sondern ein neues Modell erzeugt. Für die Spezifikation einer einzelnen Transformation wird jeweils eine Quellmodell- und eine Zielmodellstruktur (*Structures*) angegeben.

Strukturen entsprechen im wesentlichen Modellvariablen in BOTL-Regeln. Sie können Werte für Attribute aufweisen, die im Fall eines Matches im entsprechenden Objekt des Quellmodells auftreten müssen oder aber in einem Objekt des Zielmodells erzeugt werden. Findet sich in einer Quellstruktur

ein Objekt eines bestimmten Typs, so kann dieses Objekt auf alle Objekte des Quellmodells von diesem Typ oder eines seiner Subtypen matchen.

Zusätzlich können in Strukturen Kardinalitäten an Assoziationen angegeben werden, mit denen umfangreiche Strukturen kompakter dargestellt werden können. Durch die Möglichkeit auch den Wert Null als Kardinalität anzugeben können auch negative Kontexte spezifiziert werden.

Die Objekte der Quell- und Zielstruktur können miteinander in Bezug stehen. Hierfür erweitert UMLX die UML um einer Reihe zusätzlicher Arten von Beziehungen. Abbildung 6.20 gibt einen Überblick über die verschiedenen Beziehungstypen.

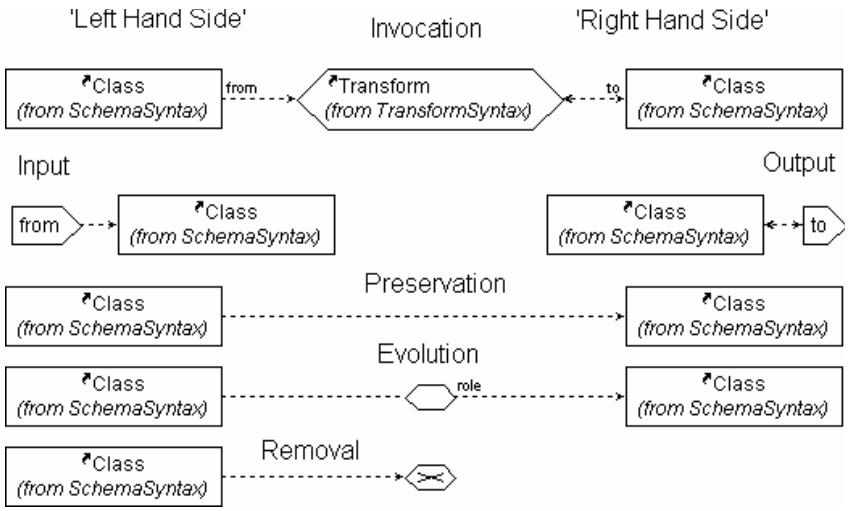


Abbildung 6.20: Graphische Elemente von UMLX, um welche die UML erweitert wird [Wil03b]

Die möglichen Arten von Transformationsbeziehungen zwischen Objekten in Strukturen sind im Einzelnen:

Preservation: Kopiert das Quellobjekt und alle komponierten Objekte in das Zielmodell.

Evolution: Erzeugt aus dem Quellobjekt ein neues Zielobjekt. Dessen Attribute können ggf. andere Werte aufweisen.

Removal: Das Quellobjekt wird gelöscht. Eine Preservation-Beziehung wird hierdurch überschrieben, d.h. komponierte Objekte werden nicht mehr kopiert.

Invocation: Eine andere Transformation wird aufgerufen, der Ein- und Ausgabekontext wird der Transformation von der aufrufenden Transformation übergeben.

Die Identitäten der erzeugten Objekte errechnet sich aus den Rollenanamen der Transformationsbeziehungen und der durch sie gematchten Quellobjekte.

Objekte der Quell- bzw. Zielstruktur können mit *Ein-* bzw. *Ausgabe-Ports* verbunden werden. Eine Transformation erhält während ihrer Ausführung konkrete Objekte des Quellmodells als Eingabekontext und produziert Zielobjekte als Ausgabekontext. Hierdurch können Transformationen hierarchisch organisiert und verknüpft werden. Abbildung 6.21 zeigt ein Beispiel für eine UMLX-Transformation, die zwei Untertransformationen aufruft.

Die UMLX-Implementierung soll die graphisch spezifizierten Transformationen schließlich in ein XSLT-Dokument umwandeln, anhand dessen sich XMI-Repräsentationen von UML-Modellen transformieren lassen. Aufgrund des sehr frühen Entwicklungsstadiums von GMT lassen sich jedoch kaum

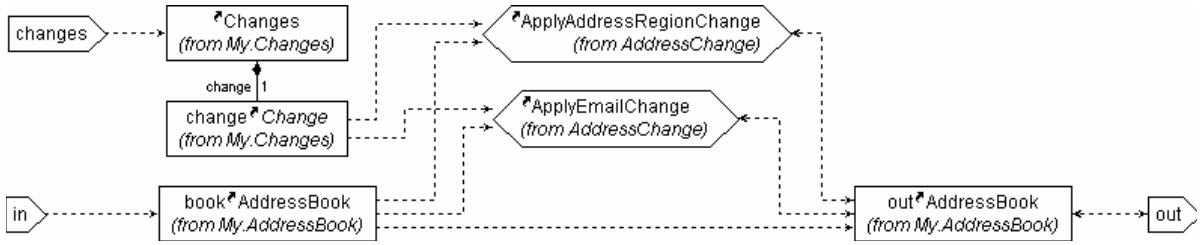


Abbildung 6.21: Beispiel für eine zusammengesetzte UMLX-Transformation [Wil03b]

Vergleiche zum erstellten BOTL-Werkzeug ziehen. Die angestrebten Fähigkeiten zur Transformation und dem Zusammenführen von Modellen überschneiden sich jedoch offensichtlich mit denen des BOTL-Werkzeuges. Jedoch beschränkt sich GMT/UMLX im Gegensatz zu BOTL ausschließlich auf XMI-Modelle zur Software-Entwicklung, bietet jedoch in der angestrebten Form auch eine für die MDA-basierte Entwicklung zielgerichtetere Unterstützung für Entwickler an.

Die Sprache UMLX selbst erscheint an vielen Stellen noch sehr unausgereift. So umfasst die „UMLX Language Definition“ [Wil03a] lediglich fünf Seiten mit zumeist leeren Kapiteln. Das von anderen Transformationswerkzeugen bekannte Konzept, Transformationen über Schnittstellen zu kombinieren, führt auch hier schnell zu unübersichtlichen Spezifikationen. Auch die Metamodellkonformität der erzeugten Modelle, sowie die Anwendbarkeit von UMLX-Spezifikationen kann nicht verifiziert werden.

6.6.5 Zusammenfassung

Allen hier vorgestellten Werkzeugansätzen ist gemein, dass sie die Transformation objektorientierter Modelle zum Ziel haben. Während PROGRES hierfür eine etwas gewöhnungsbedürftige Repräsentation aus attributierten Graphen und Graphschemata verwendet, unterstützen Optimix und GReAT auch die direkte Transformation von Objektstrukturen. GMT beschränkt sich auf die Transformation der XMI-Darstellungen von UML-Modellen, wobei der hierzu vorgeschlagene Transformationsansatz UMLX für beliebige objektorientierte Modelle verwendbar ist. Das BOTL-Werkzeug transformiert beliebige objektorientierte Modelle, die über jeweils Adaptoren ein- und ausgelesen werden können. Derzeit besteht die Möglichkeit zur Verarbeitung von Java-Objektstrukturen, Adaptoren für XMI-Dokumente werden gegenwärtig erstellt.

UMLX ermöglicht die Spezifikation von Model-to-Model-Transformationen. BOTL erlaubt hingegen die Transformation mehrerer Quellmodelle in ein Zielmodell. Alle anderen vorgestellten Ansätze spezifizieren Model-Rewrite-Transformationen, in deren Verlauf jeweils ein bestehendes Modell modifiziert wird.

Die verschiedenen Ansätze verwenden alle regelbasierte Sprachkonstrukte, die oftmals parametrisierbar sind. Zusätzlich stehen vielfach Makromechanismen und Mechanismen zur Steuerung eines Kontrollflusses zur Verfügung. PROGRES und GReAT verfügen neben einer graphischen Syntax auch über eine Vielzahl textbasierter Spezifikationstechniken, welche die Mächtigkeit der Sprache erweitern. Während BOTL und UMLX eine rein graphische Syntax aufweisen, verfügt Optimix lediglich über eine textbasierte Syntax. Lediglich BOTL beschränkt sich auf eine einzige, rein deklarative Form der Syntax zur Spezifikation von Transformationen.

Die Ausdrucksmächigkeit der meisten eingesetzten Sprachen zur Modelltransformation profitiert von den verwendeten heterogenen Sprachparadigmen und Mechanismen zur Steuerung des Ablaufs einer Transformation. Um die Konfluenz der Transformationen sicherzustellen, werden explizit Kon-

texte übergeben, so dass ein Nichtdeterminismus ausgeschlossen wird, oder zumindest ausgeschlossen werden kann. Zusätzliche Kontrollflussoperatoren zur Steuerung des Ablaufs einer Transformation erhöhen die Ausdrucksmächtigkeit der hier untersuchten Sprachen gegenüber BOTL. Allerdings wird die Lesbarkeit und Verständlichkeit von Spezifikationen durch die Verwendung dieser Sprachelemente stark beeinträchtigt.

Die Ausführbarkeit (bzw. Anwendbarkeit) von Transformationen ist lediglich in PROGRES sichergestellt. Hier führen nicht ausführbare Teile einer Transformationsspezifikation zu keiner Veränderung am Graphen. Auch der Nachweis, dass durch eine einmal spezifizierte Modelltransformation immer metamodellkonforme Modelle erzeugt werden, ist lediglich innerhalb von PROGRES möglich. Allerdings beschränkt sich dieser Nachweis auf die eingeschränkte Klasse von durch PROGRES-Graphschemata spezifizierbaren Metamodellen. BOTL erlaubt eine automatisierte Verifikation von Regelwerken bezüglich ihrer Anwendbarkeit und Metamodellkonformität.

Ansätze für bijektive oder bidirektionale Abbildungen existieren zwar für Graphgrammatiken und BOTL, werden jedoch derzeit von keinem Transformationswerkzeug direkt untertstützt.

Zusammenfassend lässt sich sagen, dass BOTL durch die Wahl einer sehr intuitiven graphischen Syntax nicht über die gleiche Ausdrucksmächtigkeit der anderen Ansätze verfügt. Dies wirkt sich jedoch in der Praxis lediglich dann negativ aus, wenn im Quellmodell nach sehr generischen Strukturmustern, wie z.B. der konvexen Hülle einer Struktur gesucht wird. Eine Erweiterung der BOTL um Makromechanismen und negative Elemente in Quellmodellvariablen, durch die Elemente im Quellmodell definiert werden, die nicht vorhanden sein dürfen, ist jedoch prinzipiell ohne weiteres möglich. Die Hinzunahme von Sprachelementen zur Steuerung des Transformationsablaufs selbst würde jedoch den rein deklarativen Charakter der Sprache untergraben.

Wird dieser Umstand akzeptiert, so ergeben sich jedoch eine Reihe von Vorteilen gegenüber den hier untersuchten Ansätzen. Zum einen lässt sich für BOTL-Transformationen formal nachweisen, ob sie anwendbar (d.h. konfluent im Sinne von Graphgrammatiken) sind und ob das Ergebnis einer Transformation konform zu einem gegebenen Zielmetamodell ist. Diese Eigenschaft ist insbesondere für die automatisierte Ausführung von Transformationen, bei denen die Quellmodelle im vornherein unbekannt sind, von besonderer Bedeutung. Zum anderen sind BOTL-Spezifikationen deutlich besser lesbar und intuitiver interpretierbar als die Spezifikationen vergleichbarer Ansätze. So wird lediglich eine einzige Notation (Regeldiagramme) verwendet, welche rein deklarativ den Zusammenhang zwischen Zielmodell und Quellmodellen spezifiziert.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Modellbasierte Entwicklungsansätze tragen in vielerlei Hinsicht zu einer Steigerung der Effizienz des Entwicklungsprozesses und der Qualität der erstellten Software-Produkte bei. So erlaubt es die Verwendung von Modellen einem Entwickler sich durch Abstraktion von unwichtigen Details auf die wesentlichen Aufgaben bei der Modellierung eines Software-Systems zu konzentrieren. Eine für einen bestimmten Typ von Anwendung angepasste Modellbildung ermöglicht, im Gegensatz zu sehr universellen Ansätzen wie der UML, eine wesentlich gezieltere Unterstützung des Entwicklungsprozesses. Die Einschränkung des Lösungsraums durch geeignete Metamodelle, die Angabe konsistenzerhaltender Modelloperationen, sowie eine automatische Verfeinerung konzeptueller Modelle tragen weiter wesentlich zur Vermeidung von Fehlern im Verlauf der Entwicklung bei.

Die Bearbeitung der konzeptuellen Modelle erfolgt anhand von Artefakten, deren Inhalte Sichten auf ein solches Modell darstellen. Um eine Überfrachtung konzeptueller Metamodelle mit Elementen aus den für die Artefakte verwendeten Beschreibungstechniken zu vermeiden, müssen geeignete Abbildungen zwischen den Beschreibungstechniken und den konzeptuellen Modellen existieren.

Ein wohldefinierter, flexibler Entwicklungsprozess auf Basis eines einheitlichen Produktmodells der Artefakttypen bietet Entwicklern die nötige Hilfestellung bei der Spezifikation und Erstellung eines Software-Systems, indem er sicherstellt, dass die Entwicklungsaktivitäten zu einer konsistenten Systemspezifikation und -implementierung führen.

Innerhalb dieser Arbeit wurden zunächst wesentliche Konzepte und Techniken für die Realisierung eines modellbasierten Entwicklungsprozesses entworfen. In diesem ist festgelegt, wie Modelle durch die Integration von Informationen aus Artefakten aufgebaut werden können und wie Modelle durch Abbildungen verfeinert werden. So ist es beispielsweise möglich zu verifizieren, ob durch die Integration von Artefakten in ein verfeinertes Modell die Verfeinerungsbeziehung zu dem abstrakten Modell erhalten bleibt.

Der im Rahmen dieser Arbeit vorgestellte Prozessmusteransatz bietet eine geeignete Modellierungssprache für Entwicklungsprozesse. Durch die formale Spezifikation von Entwicklungsschritten als Transformationen einer Instanz des Produktmodells ist es möglich, verschiedene Entwicklungsansätze zu einer speziell für ein Projekt abgestimmten Vorgehensweise zu kombinieren.

Sowohl für die Integration von Artefakten in konzeptuelle Modelle und deren Verfeinerung, als auch für die Transformation von Instanzen eines Produktmodells werden geeignete Transformationsmechanismen benötigt. Die in dieser Arbeit vorgestellte Sprache BOTL für Modelltransformationen ist in der Lage die Anforderungen, die sich für einen solchen Mechanismus ergeben, für objektorientierte Modelle zu erfüllen. BOTL verfügt über eine formale Semantik, eine intuitive graphische Syntax und erlaubt es, die Anwendbarkeit, Metamodellkonformität und Bijektivität von Transformationen formal zu verifizieren. Durch die Erweiterung der BOTL für Rewrite-Transformationen sind auch Transformationen einer Instanz eines Produktmodells, wie sie für den Prozessmusteransatz benötigt werden, möglich.

Der Einsatz von BOTL für einen modellbasierten Entwicklungsprozess wurde anhand der KOGITO-Methodik exemplarisch dargestellt. Die Möglichkeit, die Anwendbarkeit und Metamodell-

konformität von BOTL-Regelwerken für beliebige Eingaben automatisiert nachzuweisen, erlaubt es, den BOTL-Transformationsmechanismus direkt in eine Werkzeugunterstützung zu integrieren.

Mit dem in Teil 6 vorgestelltem Werkzeug wurde bereits eine lauffähige Implementierung für die Ausführung von BOTL-Transformationen geschaffen. Das Werkzeug ist in der Lage Java-Objektgeflechte zu transformieren und kann somit direkt für die Integration von Werkzeugen zur Unterstützung eines modellbasierten Entwicklungsansatzes eingesetzt werden. Auch eine automatisierte Verifikation von BOTL-Spezifikationen ist möglich. Somit wurde die Umsetzbarkeit der vorgestellten Techniken zur Transformation von objektorientierten Modellen in der Praxis verifiziert. Auch die vorgestellte exemplarische Anwendung der Techniken für die KOGITO-Methodik ließ sich mit dem Werkzeug durchführen.

7.2 Ausblick

Die in dieser Arbeit vorgestellten Konzepte und Techniken bieten bereits eine gute Basis für die Umsetzung modellbasierter Entwicklungsprozesse. Dennoch bieten sich aufbauend auf diesen Ergebnissen eine Reihe möglicher Erweiterungen an. Diese betreffen im Wesentlichen die Syntax von BOTL-Regelwerken, die Mächtigkeit der Verifikationstechniken, sowie die erstellte Werkzeugunterstützung.

In [CH03] wird bemängelt, dass die Spezifikation von Transformationen von UML-Modellen mit BOTL aufgrund der Komplexität des UML-Metamodells zu komplexen Regelwerken führt. Diesem Sachverhalt kann durch die Einführung angepasster Schreibweisen für Regeln zur Transformation solcher Metamodelle begegnet werden. Beispielsweise lassen sich in Regeln Elemente der konkreten Syntax von UML-Diagrammtypen darstellen. Die Textfelder in diesen Elementen können mit den Termen für die Identifikator- und Attributwerte der abstrakten Syntax der Notation beschriftet werden. Ein solches Vorgehen ist jedoch generell nicht für jede Art von Diagrammen möglich, da oftmals nicht sämtliche hierfür nötigen Informationen (wie z.B. Identifikatoren) in der konkreten Syntax einer Notation textuell dargestellt werden.

Eine weitere syntaktische Vereinfachung wäre es Polymorphie bei Objektvariablen zu erlauben. In diesem Fall würde eine Objektvariable auf alle Objekte derselben Klasse und alle Objekte von geerbten Klassen matchen. Ein Zugriff auf geerbte Attribute dieser Objekte für die Transformation ist jedoch hierbei nicht mehr möglich. Zudem würde hierdurch die Ausdrücksmächtigkeit der BOTL eingeschränkt werden. Diesem Sachverhalt kann jedoch durch weitere syntaktische Erweiterungen begegnet werden.

Bei der Transformation objektorientierter Modelle und der Verifikation der Metamodellkonformität werden von BOTL noch einige Aspekte, die sich in der MOF bzw. der UML finden, nicht berücksichtigt. So wird für den Nachweis der Metamodellkonformität die in der MOF geforderte Eigenschaft, dass ein Objekt lediglich in von einem anderen Objekt komponiert werden kann, noch nicht berücksichtigt. Auch die Zykelfreiheit von Kompositionsbeziehungen ist durch den BOTL-Formalismus nicht per se gewährleistet. Idealer Weise wäre eine Erweiterung der bestehenden Verifikationstechniken um die Berücksichtigung der Object Constraint Language (OCL), oder einer Untermenge der OCL, denkbar. Da die zuvor genannten Constraints innerhalb der MOF ebenfalls durch OCL-Constraints beschrieben sind, ließen sich diese dann leicht integrieren.

Generell gehen die Verifikationstechniken von idealen Funktionen und Prädikaten für Eigenschaften von Regelwerken (wie z.B. *maxRelevant*) aus, für die in der Regel pessimistische Abschätzverfahren angegeben werden (z.B. *relevant*). Durch die Berücksichtigung zusätzlicher Constraints auf den Quellmetamodellen (wie z.B. OCL-Constraints) könnten sich diese Abschätzungen weiter verbessern lassen. Auch die Verfahren zum Nachweis der Bijektivität von Regelwerken ließen sich ggf. durch

den Einsatz graphtheoretischer Verfahren weiter optimieren.

Um die Mächtigkeit von BOTL weiter zu erhöhen, ist es denkbar auch die Angabe eines negativen Kontextes in der Quellmodellvariable zuzulassen. Ein negativer Kontext macht Aussagen über die Nichtexistenz von Elementen im Quellmodell. Dies würde es beispielsweise erlauben, sämtliche Geschäftsprozesse im BRM zu erfassen für die *kein* Aktor existiert. Eine Erweiterung dieser Art erfordert jedoch eine grundlegende Überarbeitung der existierenden Verifikationstechniken.

Die Erweiterungen der Syntax und der Verifikationstechniken lassen sich auch in die bestehende Werkzeugunterstützung integrieren. Durch den Einsatz eines kommerziellen Mathematik-Framworks, welches das Lösen von Ungleichungssystem beherrscht, könnte die Menge der durch das Werkzeug abgebildeten Verifikationstechniken vervollständigt werden. Zusätzlich kann das BOTL-Werkzeug um weitere Adaptoren für technologiespezifische Modelle erweitert werden. Derzeit entstehen Adaptoren zur Ein- und Ausgabe von Modellen in einer XMI-Repräsentation, was eine Anbindung von CASE-Tools möglich macht. Neben einer Reihe von Maßnahmen zur Steigerung der Effizienz des generischen Transformators stellte auch die vollständige Generierung von Transformatoren aus den XML-Spezifikationen eine Möglichkeit dar, Transformationen noch effizienter ausführen zu können.

A Beweise

A.1 Beweise zum BOTL-Formalismus

A.1.1 Instanzierbarkeit von Metamodellen

Beweis-Skizze zu Lemma 3.2.1, Seite 67: Sei *asso* eine Klassenassoziation mit den Enden ae_0 und ae_1. Eine Gleichung EQ_{asso} des Gleichungssystems $IES(mm)$ formuliert genau die Eigenschaft, dass die Zahl der im Modell auftretenden Assoziationsenden des Typs ae_0 gleich der des Typs ae_1 sein muss. Dies entspricht der Forderung, dass sämtliche Objektassoziationen eines Modells konsistent sein müssen (siehe 3.1.14). Die Menge von Assoziationsenden eines Typs wird hierbei ermittelt, indem über alle Konfigurationen die über entsprechende Objektassziationsenden verfügen aufaddiert wird.

Für ein Modell gibt der Wert einer Unbekannten $x_{class}^{CF_{class}^i}$ an wie oft die Konfiguration CF_{class}^i der Klasse *class* im Modell vorkommt. Folglich müssen alle Unbekannten ganzzahlige, nicht-negative Lösungen haben.

Existiert nun für $x_{class}^{CF_{class}^i}$ keine endliche Lösung ungleich Null, so kommt diese Konfiguration in jeder Instanz des Modells null mal vor, d.h. es gibt kein Modell mit einem Objekt, welches die in Definition 3.2.5 geforderten Eigenschaften aufweist.

Existiert umgekehrt eine solche Lösung bei der zudem alle anderen Unbekannten endliche, nicht-negative Werte haben, so lässt sich leicht ein Beispiel für ein Modell angeben, in dem ein Objekt der geforderten Konfiguration vorkommt und welches zudem eine gültige Instanz des Metamodells darstellt. ❑

Beweis-Skizze zu Lemma 3.2.2, Seite 67: Sind die Forderungen aus Lemma 3.2.2 nicht erfüllt, so ist gemäß Lemma 3.2.1 keine Konfiguration der Klasse instanzierbar. Dementsprechend kann kein Objekt dieser Klasse in einem gültigen (endlichen) Modell existieren, was gemäß Definition 3.2.4 bedeutet, dass die Klasse nicht instanzierbar ist.

Sind die Forderungen aus Lemma 3.2.2 erfüllt, so lässt sich leicht ein Beispiel für ein gültiges Modell angeben, welches zumindest ein Objekt der entsprechenden Klasse enthält. ❑

Beweis-Skizze zu Satz 3.2.1, Seite 67: Sind die Forderungen aus Satz 3.2.1 nicht erfüllt, so ist gemäß Lemma 3.2.2 keine Klasse des Metamodells. Dementsprechend kann kein Objekt des Metamodells in einem gültigen Modell existieren, was gemäß Definition 3.2.1 bedeutet, dass das Metamodell nicht instanzierbar ist.

Sind die Forderungen aus Lemma 3.2.1 erfüllt, so lässt sich leicht ein Beispiel für ein gültiges Modell angeben, welches eine endliche Anzahl von Instanzen zumindest einer Klasse des Metamodells enthält. ❑

A.1.2 Regelwerksanwendung

Transformation von Quellmodellfragmenten in Zielmodellfragmente

Beweis von Lemma 3.5.1, Seite 90: Direkt aus den Definitionen 3.5.4, 3.5.5 und 3.5.6 ersichtlich. ❑

Beweis von Lemma 3.5.2, Seite 92:

Es wird nun untersucht, wann $ov_0 \not\sim ov_1$ zutreffen kann und für jeden Fall gezeigt, dass die beiden Objektvariablen dann kein identisches Objekt erzeugen können.

Kurzschreibweisen:

$$
\begin{aligned}
a &:= ov_0|_{otv} \neq ov_1|_{otv} \\
b &:= ov_0|_{otv}|_{Keys} = \emptyset \\
c &:= ov_0|_{oiv} \sim ov_1|_{oiv} \\
d &:= \forall (n,t) \in ov_0|_{otv}|_{Keys} : \left(ov_0.n \notin t|_{T_{val}} \vee ov_1.n \notin t|_{T_{val}} \vee ov_0.n = ov_1.n\right)
\end{aligned}
$$

Gemäß Definition 3.5.8 gilt $ov_0 \not\sim ov_1$, falls

$$
\begin{aligned}
& ov_0 \not\sim ov_1 \\
\Leftrightarrow\ & \neg(a \wedge (b \wedge c \vee \neg b \wedge d)) \\
\Leftrightarrow\ & \neg a \vee \neg(b \wedge c \vee \neg b \wedge d)) \\
\Leftrightarrow\ & \neg a \vee \neg(b \wedge c) \wedge \neg(\neg b \wedge d) \\
\Leftrightarrow\ & \neg a \vee (\neg b \vee \neg c) \wedge (b \vee \neg d) \\
\Leftrightarrow\ & \underbrace{\neg a}_{\text{Fall 1}} \vee \underbrace{\neg b \wedge b}_{\text{= false}} \vee \underbrace{b \wedge \neg c}_{\text{Fall 2}} \vee \underbrace{\neg b \wedge \neg d}_{\text{Fall 3}} \vee \underbrace{\neg c \wedge \neg d}_{\text{Fall 4}}
\end{aligned}
$$

Fall 1: $ov_0|_{otv} \neq ov_1|_{otv}$

Dieser Fall ist trivial, da die Objektvariablen immer Objekte unterschiedlichen Typs, und somit unterschiedliche Objekte, erzeugen.

Fall 2: $ov_0|_{otv}|_{Keys} = \emptyset \wedge ov_0|_{oiv} \not\sim ov_1|_{oiv}$

Für die Aussagen aus Definition 3.5.7 werden die folgenden Kurzschreibweisen eingeführt:

$$
\begin{aligned}
A &:= oiv_0, oiv_1 \in \mathbb{ID} \cup \mathbb{VAR} \\
B &:= \neg(\mathbb{ID} \ni oiv_0 \neq oiv_1 \in \mathbb{ID}) \\
C &:= oiv_0 = (s_0, \ldots, s_n), oiv_1 = (t_0, \ldots, t_n) \ \wedge \\
&\quad \nexists i \in \{0, \ldots, n\} : \mathbb{ID} \ni s_i = t_i \in \mathbb{ID}
\end{aligned}
$$

Wegen $ov_0|_{otv}|_{Keys} = \emptyset$ existieren keine Primärschlüssel. Sei $oiv_0 := ov_i|_{oiv}$ und $oiv_1 := ov_j|_{oiv}$. Dann lässt sich die Aussage

$$ov_i|_{oiv} \not\sim ov_j|_{oiv}$$

umformen in:

$$(\neg A \wedge \neg C) \vee (\neg B \wedge \neg C)$$

Fall 2.1: $\neg A \wedge \neg C$
$\neg A$ bedeutet mindestens ein Elements aus $\{oiv_0, oiv_1\}$ ist $\diamondsuit$ oder ein Tupel.
Fall 2.1.1: oiv_0 oder oiv_1 hat den Wert $\diamondsuit$.
In diesem Fall gilt offensichtlich auch $\neg C$ und

$$\overset{\text{Def. 3.5.4 (3.25), } genId}{\Rightarrow} ov_i \text{ und } ov_j \text{ erzeugen Objekte mit unterschiedlichem Objektidentifikator.}$$

Fall 2.1.2: Ist oiv_0 oder oiv_1 ein Tupel und *negC* gilt, so bedeutet dies

$$\neg(oiv_0 = (s_0,\ldots,s_n), oiv_1 = (t_0,\ldots,t_n)) \quad \vee \tag{A.1}$$
$$\exists i \in \{0,\ldots,n\} : \mathbb{ID} \ni s_i = t_i \in \mathbb{ID} \tag{A.2}$$

In diesem Fall führen oiv_0 und oiv_1 zu Objekten mit unterschiedlichen Identifikatoren, da die Identifikatoren entweder Tupel unterschiedlicher Länge sind (A.1) oder sich in einem konstanten Wert an derselben Stelle unterscheiden (A.2).
Fall 2.2: $\neg B \wedge \neg C$
Offensichtlich gilt:

$$\neg B \Rightarrow \neg C$$

Gilt $\neg B$, so bedeutet dies:

$$oiv_0 \in \mathbb{ID} \wedge oiv_1 \in \mathbb{ID} \wedge oiv_0 \neq oiv_1$$

D.h. die Objektidentifikatoren werden aus unterschiedlichen konstanten Identifikatorwerten erzeugt und sind somit unterschiedlich.

Fall 3: $ov_0|_{otv}|_{Keys} \neq \emptyset \wedge \exists (n,t) \in ov_0|_{otv}|_{Keys} : \left(ov_0.n \in t|_{T_{val}} \wedge ov_1.n \in t|_{T_{val}} \wedge ov_0.n \neq ov_1.n\right)$
Wegen $ov_0|_{otv}|_{Keys} \neq \emptyset$ verfügt die Klasse der Objektvariablen über Primärschlüssel. Weiterhin gibt es in beiden Objektvariablen eine Objektattributvariable gleichen Typs die konstant ist, jedoch unterscheiden sich die beiden Werte $ov_0.n \neq ov_1.n$. Folglich existieren unterschiedliche Primärschlüssel, was bedeutet, dass die beiden Objektvariablen immer unterschiedliche Objekte erzeugen.

Fall 4: $ov_0|_{oiv} \not\sim ov_1|_{oiv} \wedge \exists (n,t) \in ov_0|_{otv}|_{Keys} : \left(ov_0.n \in t|_{T_{val}} \wedge ov_1.n \in t|_{T_{val}} \wedge ov_0.n \neq ov_1.n\right)$
Für diesen Fall gibt es zwei Möglichkeiten:
Fall 4.1: $ov_0|_{otv}|_{Keys} = \emptyset$
$\Rightarrow$ Beweis wie für Fall 2.
Fall 4.1: $ov_0|_{otv}|_{Keys} \neq \emptyset$
$\Rightarrow$ Beweis wie für Fall 3. ❑

Zusammenführen der Zielmodellfragmente

Beweis-Skizze zu Lemma 3.5.3, Seite 93:

$$mergeable(OB_0, OB_1) = mergeable(OB_1, OB_0)$$

Lemma 3.5.3 kann leicht durch Einsetzen in die Definition von *mergeable* (Def. 3.5.10) bewiesen werden. ❑

Beweis-Skizze zu Lemma 3.5.4, Seite 93: (i) und (ii) sind direkt aus Definition 3.5.10 von *mergeable* ersichtlich. Gemäß modus tollens folgt (iii) direkt aus (i). ❑

Beweis-Skizze zu Lemma 3.5.5, Seite 94: Zu zeigen:

$$OBmerge(OB_0, OB_1) = OBmerge(OB_1, OB_0)$$

Lemma 3.5.5 kann leicht durch Einsetzen in die Definition von *OBmerge* (Def. 3.5.11) bewiesen werden. ❑

Beweis von Lemma 3.5.6, Seite 95:

$$\begin{array}{rl} & OBmerge(OB'_0, OB'_1) = \bot \\ \Rightarrow & \neg mergeable(OB'_0, OB'_1) \\ \overset{\text{Lemma3.5.4}(iii)}{\Rightarrow} & \neg mergeable(OB_0, OB_1) \\ \Rightarrow & OBmerge(OB_0, OB_1) = \bot \end{array}$$

❑

Beweis von Lemma 3.5.7, Seite 95: Für die drei möglichen Fälle

$$(i)\ \ o|_{oi} \in \{o \in OB_0 : o|_{ot} = c\}|_{oi} \ \wedge\ o|_{oi} \notin \{o \in OB_1 : o|_{ot} = c\}|_{oi}$$

$$(ii)\ \ o|_{oi} \notin \{o \in OB_0 : o|_{ot} = c\}|_{oi} \ \wedge\ o|_{oi} \in \{o \in OB_1 : o|_{ot} = c\}|_{oi}$$

$$(iii)\ \ o|_{oi} \in \{o \in OB_0 : o|_{ot} = c\}|_{oi} \ \wedge\ o|_{oi} \in \{o \in OB_1 : o|_{ot} = c\}|_{oi}$$

ist die Aussage direkt aus den entsprechenden Fällen von Definition 3.5.11 ableitbar. ❑

Beweis von Lemma 3.5.8, Seite 95: (iii) gilt offensichtlich da *OBmerge* strikt ist. $(i) \Leftrightarrow (ii)$ da *OBmerge* kommutativ ist, d.h. $OBmerge(OB_0, OB_1) = OBmerge(OB_1, OB_0)$

o.B.d.A. wird (ii) bewiesen:

Beweis durch Widerspruch:

$$\begin{array}{rl} & \neg mergeable(OB_0, OB_2) \\ \overset{\text{Lemma 3.5.4}}{\Rightarrow} & \exists o', o'' \text{ mit } \neg mergeable(\{o'\}, \{o''\}) \wedge o' \in OB_0 \wedge o'' \in OB_2 \\ \Rightarrow & o'|_{oi} = o''|_{oi} \end{array}$$

Fall 1:

$$\begin{array}{rl} & o'|_{ot} \neq o''|_{ot} \\ \overset{\text{Lemma 3.5.6}}{\Rightarrow} & \exists o_m \in OBmerge(OB_1, \{o''\}) \text{ mit } \ o_m|_{oi} = o''|_{oi} \wedge o_m|_{ot} = o''|_{ot} \\ & \qquad\qquad \wedge o_m|_{oi} = o'|_{oi} \wedge o_m|_{ot} = o'|_{ot} \\ \Rightarrow & \neg mergeable(\{o'\}, \{o_m\}) \\ \Rightarrow & OBmerge(\{o'\}, \{o''\}) = \bot \\ \overset{\text{Lemma 3.5.6}}{\Rightarrow} & OBmerge(OB_0, OBmerge(OB_1, OB_2)) = \bot \\ & \lightning \text{ (Widerspruch)} \end{array}$$

Fall 2: $\exists(a,v') \in o'|_V \wedge \exists(a,v'') \in o''|_V$ mit $v' \neq v'' \wedge v' \neq \Diamond \wedge v'' \neq \Diamond$

$\overset{\text{Lemma 3.5.4}}{\Rightarrow}$ $\exists o_m \in OBmerge(OB_1, \{o''\})$ mit $o_m|_{oi} = o''|_{oi} \wedge (a,v'') \in o_m|_V$

$\Rightarrow$ $OBmerge(\{o'\}, \{o_m\}) = \bot$

$\overset{\text{Lemma 3.5.6}}{\Rightarrow}$ $OBmerge(OB_0, OBmerge(OB_1, OB_2)) = \bot$

$\lightning$ (Widerspruch)

$\Rightarrow (ii), (i)$

❑

Beweis von Lemma 3.5.9, Seite 95: Zu zeigen:

$$\underbrace{OBmerge(OB_0, OBmerge(OB_1, OB_2))}_{=:l.s.} = \underbrace{OBmerge(OBmerge(OB_0, OB_1), OB_2)}_{=:r.s.}$$

Gemäß Lemma 3.5.8 gilt: $r.s. = \bot = l.s. \vee r.s. \neq \bot \neq l.s.$

Fall 1: $r.s. = \bot \wedge l.s. = \bot$
$\Rightarrow l.s. = r.s.$

Fall 2: $r.s. \neq \bot \wedge l.s. \neq \bot$

Beweis durch Widerspruch: Es gilt (I) oder (II)

$$\exists c' \in mm|_{CB} : \{o \in l.s. : o|_{otv} = c'\}|_{oi} \neq \{o \in r.s. : o|_{otv} = c'\}|_{oi} \tag{I}$$

$$\begin{aligned} &\exists o' \in l.s.|_{OB}, o'' \in r.s.|_{OB} : \\ &\quad o'|_{oi} = o''|_{oi} \wedge o'|_{ot} = o''|_{ot} \wedge o'|_V \neq o''|_V) \end{aligned} \tag{II}$$

Fall I: $\exists c \in \mathbb{C}, o|_{oi} \in \{o \in O^0 : o|_{ot} = c\}|_{oi} \cup \{o \in O^1 : o|_{ot} = c\}|_{oi} \cup \{o \in O^2 : o|_{ot} = c\}|_{oi} :$
$o|_{oi} \notin \{o \in l.s. : o|_{ot} = c\}|_{oi} \cup \{o \in r.s. : o|_{ot} = c\}|_{oi}$

$\overset{\text{Lemma 3.5.7}}{\Rightarrow}$ $\lightning$ (Widerspruch)

Fall II: Gemäß Definition 3.5.11 gilt:

$$\begin{aligned} &o'|_{ot} = o''|_{ot} \wedge o'|_V|_a = o''|_V|_a \\ \Rightarrow\ &\exists(a,v') \in o'|_V \wedge \exists(a,v'') \in o''|_V \text{ mit } v' \neq v'' \end{aligned}$$

Gemäß Lemma 3.5.4 gilt:

$$\begin{aligned} & mergeable(\{o'\}, \{o''\}) \\ \overset{v' \neq v''}{\Rightarrow}\ & (v' = \Diamond \wedge v'' \neq \Diamond) \vee (v' \neq \Diamond \wedge v'' = \Diamond) \\ \overset{\text{Def. OBmerge}}{\Rightarrow}\ & \lightning \text{ (Widerspruch)} \end{aligned}$$

❑

Beweis-Skizze zu Lemma 3.5.10, Seite 97: Der Beweis von Lemma 3.5.10 ergibt sich direkt aus Definition 3.5.12. ❑

Beweis von Satz 3.5.1, Seite 97: Zu zeigen:

$$mf_0 \cup_{\mathfrak{m}} mf_1 = mf_1 \cup_{\mathfrak{m}} mf_0$$

OBmerge ist gemäß Lemma 3.5.5 kommutativ. Somit lässt sich durch Substitution leicht zeigen, dass Satz 3.5.1 gilt. ❑

Beweis von Satz 3.5.2, Seite 97: Zu zeigen:

$$(mf_0 \cup_{\mathfrak{m}} mf_1) \cup_{\mathfrak{m}} mf_2 = mf_0 \cup_{\mathfrak{m}} (mf_1 \cup_{\mathfrak{m}} mf_2)$$

Für den Beweis von Satz 3.5.2 lassen sich drei Fälle unterscheiden, wobei

$$l.s. := (mf_0 \cup_{\mathfrak{m}} mf_1) \cup_{\mathfrak{m}} mf_2$$
$$r.s. := mf_0 \cup_{\mathfrak{m}} (mf_1 \cup_{\mathfrak{m}} mf_2)$$

Fall 1: $l.s. = \bot \wedge r.s. = \bot \Rightarrow l.s. = r.s.$

Fall 2: o.B.d.A. (gemäß Satz 3.5.1) sei $l.s. = \bot \wedge r.s. \neq \bot$

$$\begin{aligned} \bot \neq r.s.|_{OB} &= (mf_0 \cup_{\mathfrak{m}} (mf_1 \cup_{\mathfrak{m}} mf_2))|_{OB} \\ &= OBmerge(mf_0|_{OB}, OBmerge(mf_1|_{OB}, mf_2|_{OB})) \\ &= OBmerge(OBmerge(mf_0|_{OB}, mf_1|_{OB}), mf_2|_{OB}) \\ &= ((mf_0 \cup_{\mathfrak{m}} mf_1) \cup_{\mathfrak{m}} mf_2)|_{OB} \\ &= l.s.|_{OB} \end{aligned}$$

Dies widerspricht der Annahme $l.s. = \bot$.

Fall 3: $l.s. \neq \bot \wedge r.s. \neq \bot$

$$\begin{aligned} r.s.|_{OB} &= (mf_0 \cup_{\mathfrak{m}} (mf_1 \cup_{\mathfrak{m}} mf_2))|_{OB} \\ &= OBmerge(mf_0|_{OB}, OBmerge(mf_1|_{OB}, mf_2|_{OB})) \\ &= OBmerge(OBmerge(mf_0|_{OB}, mf_1|_{OB}), mf_2|_{OB}) \\ &= ((mf_0 \cup_{\mathfrak{m}} mf_1) \cup_{\mathfrak{m}} mf_2)|_{OB} \\ &= l.s.|_{OB} \end{aligned}$$

Gemäß Definition 3.5.12 enthält $r.s.|_{OA}$ bzw. $l.s.|_{OA}$ alle Objektassoziationen die in mf_0, mf_1, oder mf_2 vorkommen. Falls eine Objektassoziation in mehreren Modellfragmenten mf_i vorkommt, dann ist genau die mit der höchsten Kardinalität *card* in $l.s.|_{OA}$ enthalten (Assoziativität der binären *max*- bzw. *min*-Funktion).

Somit gilt: $l.s. = r.s.$ ❑

Beweis von Satz 3.5.3, Seite 97: Satz 3.5.3 gilt da $\cup_{\mathfrak{m}}$ kommutativ uns assoziativ ist. ❑

Regel- und Regelwerksanwendung

Beweis-Skizze zu Satz 3.5.4, Seite 98: Satz 3.5.4 folgt direkt aus Satz 3.5.3. ❑

Beweis-Skizze zu Satz 3.5.5, Seite 99: Satz 3.5.5 ist eine direkte Konsequenz aus Satz 3.5.3. ❑

Beweis von Satz 3.5.6, Seite 101:

$$
\begin{aligned}
& transform(M,R_0) \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} transform(M,R_n) \\
\overset{\text{Def. 3.5.16}}{=}\ & \bigcup_{\substack{\mathfrak{m} \\ r \in R_0}} apply(srcModel(M,r),r) \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} \bigcup_{\substack{\mathfrak{m} \\ r \in R_n}} apply(srcModel(M,r),r) \\
\overset{\text{Satz 3.5.3}}{=}\ & \bigcup_{\substack{\mathfrak{m} \\ r \in R_0 \cup \cdots \cup R_n}} apply((srcModel(M,r),r) \\
\overset{\text{Def. 3.5.16}}{=}\ & transform(M, R_0 \cup \cdots \cup R_n)
\end{aligned}
$$

❑

Beweis von Satz 3.5.7, Seite 101: *Beweis durch Induktion*

Induktionsanfang: Der Satz wir zunächst für zwei Quellmodelle m_0 und m_1 bewiesen:

Sei MF_0 die Menge der Modellfragmente, die im Verlauf der Regelwerksanwendung $transform(m_0,R)$ erzeugt werden. Gemäß Definition 3.5.5 (mft) existiert für jedes erzeugte Modellfragment $mf_i \in MF_0$ genau ein Match mfm in m_0 aus dem sich mf_i deterministisch errechnet.

Folglich muss derselbe Match mfm auch in $m_0 \cup_{\mathfrak{m}} m_1$ existieren. Da der Match im Verlauf der Transformation $transform(m_0 \cup_{\mathfrak{m}} \cup_{\mathfrak{m}} m_1, R)$ durch das selbe Regelwerk R transformiert wird, gilt für die Menge MF der im Verlauf dieser Transformation erzeugen Modellfragmente:

$$MF_0 \trianglelefteq MF$$

Vollkommen analog gilt für die Menge der durch die Transformation $transform(m_1,R)$ erzeugten Modellfragmente:

$$
\begin{aligned}
& MF_1 \trianglelefteq MF \\
\Rightarrow\ & MF_0 \cup MF_1 \trianglelefteq MF \\
\overset{\text{Ann.: } mf \neq \bot}{\Rightarrow}\ & \bigcup_{\substack{\mathfrak{m} \\ mf \in MF_0}} \cup_{\mathfrak{m}} \bigcup_{\substack{\mathfrak{m} \\ mf \in MF_1}} \trianglelefteq \bigcup_{\substack{\mathfrak{m} \\ mf \in MF}} \\
\overset{\text{Def. 3.5.16, 3.5.14}}{\Rightarrow}\ & transform(m_0,R) \cup_{\mathfrak{m}} transform(m_1,R) \trianglelefteq transform(m_0 \cup_{\mathfrak{m}} m_1, R) \qquad \text{(A.3)}
\end{aligned}
$$

Induktionsschritt: $m_0 \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} m_n \neq \bot$ vorausgesetzt gelte die Aussage für $n-1$ Quellmodelle:

$$
\begin{aligned}
& \underbrace{transform(m_0,R) \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} transform(m_{n-1},R)}_{\trianglelefteq transform(m_0 \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} m_{n-1}, R)} \cup_{\mathfrak{m}} transform(m_n,R) \\
\overset{a \trianglelefteq b \Rightarrow a \cup_{\mathfrak{m}} c \trianglelefteq b \cup_{\mathfrak{m}} c}{\trianglelefteq}\ & transform(m_0 \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} m_{n-1}, R) \cup_{\mathfrak{m}} transform(m_n,R) \\
\trianglelefteq\ & transform(m_0 \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} m_n, R) \cup_{\mathfrak{m}} transform(m_n,R) \\
\overset{\text{(A.3)}}{\trianglelefteq}\ & transform(m_0 \cup_{\mathfrak{m}} \ldots \cup_{\mathfrak{m}} m_n, R)
\end{aligned}
$$

❑

A.1.3 Rewrite-Transformationen

Beweis von Lemma 3.6.1, Seite 103: Sei r_i eine C1-Regel. Dann gilt gemäß Definition 3.6.2:

$$\begin{aligned}
&\forall ova \in r_i|_{mv_1}|_{OVA} : (ova|_{cardv} = 1 \ \wedge \ \left|ova|_{OVAE}\right| = 2) \quad \wedge \\
&\forall ov \in r_i|_{mv_1}|_{OVB} : (\exists ova \in r_i|_{mv_1}|_{OVA} : ov \in ova|_{OVAE}|_{ov}) \\
\overset{\text{Def. 3.5.5, Def. 3.5.4}}{\Rightarrow} \ &\forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}} : \\
&\quad \forall mf_j \in mft(mfm, r_i) \text{ mit } mfm \in MFM(m, r_i|_{mv_0}) : \\
&\quad\quad \forall oa \in mf_j|_{OA} : (oa|_{card} = 1 \ \wedge \ \left|oa|_{OAE}\right| = 2) \quad \wedge \\
&\quad\quad \forall o \in mf_j|_{OB} : (\exists oa \in mf_j|_{OA} : o \in oa|_{OAE}|_o) \\
\overset{\text{Merge-Lemma 3.5.10}}{\Rightarrow} \ &\forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}} : \\
&\quad mf := \bigcup_{\substack{mf_i \in mft(mfm, r_i) \\ \text{mit } mfm \in MFM(m, r_i|_{mv_0})}}^{\mathfrak{m}} mf_i \quad \wedge \\
&\quad\quad \forall oa \in mf|_{OA} : (oa|_{card} = 1 \ \wedge \ \left|oa|_{OAE}\right| = 2) \quad \wedge \\
&\quad\quad \forall o \in mf|_{OB} : (\exists oa \in mf_j|_{OA} : o \in oa|_{OAE}|_o) \\
\overset{\text{Def. } apply \text{ 3.5.14}}{\Rightarrow} \ &\forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}} : \\
&\quad mf := apply(m, r_i) \quad \wedge \\
&\quad\quad \forall oa \in mf|_{OA} : (oa|_{card} = 1 \ \wedge \ \left|oa|_{OAE}\right| = 2) \quad \wedge \\
&\quad\quad \forall o \in mf|_{OB} : (\exists oa \in mf_j|_{OA} : o \in oa|_{OAE}|_o) \\
\overset{\text{Def. C1-Modell 3.6.1}}{\Leftrightarrow} \ &\forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}} : C1(apply(m, r_i))
\end{aligned}$$

❑

A.2 Beweise zu Eigenschaften von BOTL-Regelwerken

A.2.1 Anwendbarkeit

Erzeugen gültiger Modellfragmente

Beweis von Lemma 4.1.1, Seite 112: Gemäß Definition 3.5.5 gilt:

$$
\begin{aligned}
& mft(mfm_i, r_j) = \bot \Leftrightarrow \\
& \neg\big((\exists_1 mf_i : mfr(mfm_i, r_j, mf_i) \ \wedge mf_i \in \mathbb{MF}|_{r_j|_{mv_1}|_{mm}}) \vee \\
& \quad \nexists\, mf_i : mfr(mfm_i, r_j, mf_i)\big) \\
& \Leftrightarrow \\
& (\nexists_1 mf_i : mfr(mfm_i, r_j, mf_i) \ \vee mf_i \notin \mathbb{MF}|_{r_j|_{mv_1}|_{mm}}) \wedge \quad \exists\, mf_i : mfr(mfm_i, r_j, mf_i) \\
& \overset{\text{Distributivgesetz}}{\Leftrightarrow} \\
& (\nexists_1 mf_i : mfr(mfm_i, r_j, mf_i) \wedge \exists\, mf_i : mfr(mfm_i, r_j, mf_i)) \vee \\
& (mf_i \notin \mathbb{MF}|_{r_j|_{mv_1}|_{mm}} \wedge \exists\, mf_i : mfr(mfm_i, r_j, mf_i)) \\
& \Leftrightarrow \\
& |\{mf_i : mfr(mfm_i, r_j, mf_i)\}| > 1 \vee \\
& \exists\, mf_i : mfr(mfm_i, r_j, mf_i)) \wedge mf_i \notin \mathbb{MF}|_{r_j|_{mv_1}|_{mm}}
\end{aligned}
$$

❑

Beweis von Satz 4.1.1, Seite 113: Beweis durch Widerspurch. Es gelten $\neg createsValidFragments(r)$

$$
\begin{aligned}
\overset{\text{Def. 4.1.3}}{\Leftrightarrow} \quad & \exists m \in \mathbb{M}|_r|_{mv_0}|_{mm} : \\
& \quad \exists mfm \in MFM(m, r|_{mv_0}) : mft(mfm, r) = \bot \\
\overset{\text{Lemma 4.1.1}}{\Leftrightarrow} \quad & \exists m \in \mathbb{M}|_r|_{mv_0}|_{mm} : \\
& \quad \exists mfm \in MFM(m, r|_{mv_0}) : \\
& \quad\quad |\{mf_i : mfr(mfm, r, mf_i)\}| > 1 \ \vee \qquad \text{(A.4)} \\
& \quad\quad \exists\, mf_i : mfr(mfm, r, mf_i) \wedge mf_i \notin \mathbb{MF}|_{r|_{mv_1}|_{mm}} \qquad \text{(A.5)}
\end{aligned}
$$

(A.4) steht in direktem Widerspruch zu Satz 4.1.1 (i).
(A.5) impliziert, dass $mf_i \notin \mathbb{MF}|_{r|_{mv_1}|_{mm}}$ gilt, d.h. $\neg consistent(mf_i, r|_{mv_1}|_{mm})$. Gemäß Definition 3.5.1 existieren zwei Möglichkeiten um die Konsistenz eines Modellfragments mf zu verletzen:

(i) $\neg consistent(mf|_{OB}, mm) \vee$

(ii) $\exists oa \in mf|_{OA} : consistent(oa, OB)$

Fall (ii) ist durch die Definition der Modellfragmentrelation (Def. 3.5.4) und aufgrund der Tatsache, dass Objektvariablenassoziationen einer Modellvariable konsistent zum Metamodell der Modellvariable sein müssen (siehe Def. 3.3.8, S. 74), ausgeschlossen.

Fall (i) kann auftreten falls $mf|_{OB}$ keine gültige Objektbelegung gemäß Definition 3.1.12 ist. Dann gilt:

$$
\begin{aligned}
& & & mf|_{OB} \notin OB \\
& \overset{\text{Def. 3.1.12 (3.12)}}{\Leftrightarrow} & & \exists o_i, o_j \in mf|_{OB} : \neg(o_i|_{oi} = o_j|_{oi} \Rightarrow o_i = o_j \vee o_i|_{ot} \neq o_j|_{ot}) \\
& \Leftrightarrow & & \exists o_i, o_j \in mf|_{OB} : o_i|_{oi} = o_j|_{oi} \wedge o_i \neq o_j \wedge o_i|_{ot} = o_j|_{ot}) \\
& \overset{ov_i|otv=ov_j|otv \text{ gem. Satz 4.1.1 } (ii)}{\Leftrightarrow} & & UGL_{validSrc} \wedge mfm^1_\mu|_{match_o}(ov_i)|_{oi} = mfm^1_\mu|_{match_o}(ov_j)|_{oi} \\
& & & \text{hat keine Lösung, da } UGL_{validSrc} \text{ eine Lösung hat die } mf \text{ erzeugt und} \\
& & & mfm^1_\mu|_{match_o}(ov_i)|_{oi} = mfm^1_\mu|_{match_o}(ov_j)|_{oi} \equiv o_i|_{oi} = o_j|_{oi} \text{ gilt.} \\
& \Rightarrow & & \textit{Widerspruch!}
\end{aligned}
$$

❑

Beweis von Satz 4.1.2, Seite 115: Aussage (i) entspricht genau Aussage (i) aus Satz 4.1.1.
Aussage (ii) impliziert, dass gilt (Lemma 3.5.2):

$$mfm^1_\mu|_{match_o}(ov_i)|_{oi} \neq mfm^1_\mu|_{match_o}(ov_j)|_{oi}$$

Folglich kann das System aus Satz 4.1.1 (ii) niemals eine Lösung haben.
$\overset{\text{Satz4.1.1}}{\Rightarrow} createsValidFragments(r)$ ❑

Deterministische Objektvariable

Beweis-Skizze zu Lemma 4.1.2, Seite 117: Substitution in Definition 4.1.8 liefert eine Aussage der Art $A \Rightarrow A$ die offensichtlich wahr ist. ❑

Beweis-Skizze zu Lemma 4.1.3, Seite 117: Einsetzen in Definition 4.1.8 führt zu Aussagen der Art $A \Rightarrow B \wedge B \Rightarrow C$. Da in diesem Fall auch gilt $A \Rightarrow C$ gilt auch Lemma 4.1.3. ❑

Beweis von Satz 4.1.3, Seite 118: Beweis durch Induktion über die Länge l der Folge:
Induktionsannahme: $l = 1$
Da die Folge die Länge 1 hat sind $ovae_0|_{ov}$ und $\overline{ovae}_0|_{ov}$ gemäß (4.8), (4.9) und (4.10) durch genau eine Objektvariablenassoziation verbunden. (4.12) stellt sicher, dass die entsprechende Klassenassoziation von der Klasse von $ovae_0|_{ov}$ zu der von $\overline{ovae}_0|_{ov}$ eine Multiplizität von $(1,1)$ hat. Folglich gilt $ovae_0|_{ov} \overset{mv_0}{\rightsquigarrow} \overline{ovae}_0|_{ov}$ für $l = 1$ (siehe auch Definition 3.1.14 (3.15) auf Seite 61).
Induktionsschritt: $l+1$
Gemäß der Induktionsannahme gilt die Aussage für Folgen der Länge l. Sei

$$(ovae_0, \overline{ovae}_0), \ldots, (ovae_l, \overline{ovae}_l)$$

eine Folge der Länge $l+1$. Aufgrund der Induktionsannahme kann davon ausgegangen werden, dass gilt $ovae_0|_{ov} \overset{mv_0}{\rightsquigarrow} \overline{ovae}_{l-1}|_{ov}$). Gemäß einer äquivalenten Argumentation wie bei der Induktionsannahme kann festgestellt werden, dass gilt $ovae_l|_{ov} \overset{mv_0}{\rightsquigarrow} \overline{ovae}_l|_{ov}$.
Mit Blick auf (4.11) ist bekannt, dass gilt: $\overline{ovae}_{l-1}|_{ov} = ovae_l|_{ov}$
Zusammen mit Lemma 4.1.3 kann festgestellt werden, dass gilt:

$$ovae_0|_{ov} \overset{mv_0}{\rightsquigarrow} \overline{ovae}_l|_{ov}$$

❑

Beweis-Skizze zu Satz 4.1.4, Seite 119: Gemäß Satz 4.1.4 gibt es drei Möglichkeiten um sicherzustellen, dass eine Objektvariable deterministisch ist:

Fall (4.13) impliziert gemäß Definition 3.5.4 (3.25), dass Objekte die von der Objektvariablen *ov* erzeugt werden bei jeder Modellfragmenttransformation einen eindeutigen Identifikator zugewiesen bekommen. Dementsprechend kann es keine zwei Objekte o_i, o_j wie in Definition 4.1.5 vorgesehen geben.

Fall (4.14) legt den Fall fest, in dem sämtliche Attributvariablen die zu keinen Primärschlüsselattributen gehören entweder den Wert $\diamondsuit$ oder einen konstanten Wert korrekten Typs enthalten. Dieser Wert hängt bei der Erzeugung eines neuen Objektes nicht vom Quellmodell ab und ist somit invariant.

Fall (4.15) skizziert den Fall, in dem die zur Berechnung jedes Attributs benötigten Quellobjekte bereits durch die zur Berechnung der Identität des jeweiligen Objektes benötigten Quellobjekte festgelegt sind. Dementsprechend errechnet sich bei gleichem Identifikator eines erzeugten Objektes der Wert aller Attribute immer aus denselben im Quellmodell gefundenen Werten, d.h. es werden stets dieselben Attributwerte errechnet.

Es bleibt anzumerken, dass Primärschlüsselattribute hierbei nicht berücksichtigt werden müssen, da gemäß Definition 3.5.4 (3.25) Objekte mit unterschiedlichen Werten in den Primärschlüsselattributen immer eine unterschiedliche Identität haben. ❑

Konfliktfreie Objektvariable

Beweis von Lemma 4.1.4, Seite 121:

$$
\begin{aligned}
& \neg potConflicting(R, ov_i, ov_j) \\
\overset{\text{Def. 4.1.10}}{\Leftrightarrow}\ & \nexists M \in M_R^0, o_i, o_j \text{ mit } createsObj(o_i, ov_i, r_i, srcModel(M, r_i)) \wedge \\
& \qquad createsObj(o_j, ov_j, r_j, srcModel(M, r_j)) : \\
& \qquad\quad o_i|_{oi} = o_j|_{oi} \wedge o_i|_{ot} = o_j|_{ot} \\
\overset{\nexists x:P(x) \Rightarrow \nexists x:P(x) \wedge Q(x)}{\Rightarrow}\ & \nexists M \in M_R^0, o_i, o_j \text{ mit } createsObj(o_i, ov_i, r_i, srcModel(M, r_i)) \wedge \\
& \qquad createsObj(o_j, ov_j, r_j, srcModel(M, r_j)) : \\
& \qquad\quad o_i|_{oi} = o_j|_{oi} \wedge o_i|_{ot} = o_j|_{ot} \wedge \\
& \qquad\quad \big(\exists (a_0, v_0) \in o_0|_V, (a_1, v_1) \in o_1|_V : \\
& \qquad\qquad a_0 = a_1 \wedge \neg attMergeable((a_0, v_0), (a_1, v_1))\big) \\
\overset{\text{Def. 3.5.10}}{\Leftrightarrow}\ & \nexists M \in M_R^0, o_i, o_j \text{ mit } createsObj(o_i, ov_i, r_i, srcModel(M, r_i)) \wedge \\
& \qquad createsObj(o_j, ov_j, r_j, srcModel(M, r_j)) : \\
& \qquad\quad \neg mergeable(\{o_i\}, \{o_j\}) \\
\overset{\nexists x:P(x) \Leftrightarrow \forall x:\neg P(x)}{\Leftrightarrow}\ & \forall M \in M_R^0, o_i, o_j \text{ mit } createsObj(o_i, ov_i, r_i, srcModel(M, r_i)) \wedge \\
& \qquad createsObj(o_j, ov_j, r_j, srcModel(M, r_j)) : \\
& \qquad\quad mergeable(\{o_i\}, \{o_j\}) \\
\overset{\text{Def. 4.1.9}}{\Leftrightarrow}\ & conflictFree(R, ov_i, ov_j)
\end{aligned}
$$

❑

Beweis von Lemma 4.1.5, Seite 122:

$$
\begin{aligned}
(4.18) \overset{\text{Lemma 3.5.2}}{\Rightarrow} & \nexists M \in M_R^0, o_i, o_j \text{ mit } createsObj(o_i, ov_i, r_i, srcModel(M, r_i)) \wedge \\
& \qquad createsObj(o_j, ov_j, r_j, srcModel(M, r_j)) \ : \\
& \qquad o_i|_{ot} = o_j|_{ot} \wedge o_i|_{oi} = o_j|_{oi} \\
\Rightarrow & \neg potConflicting(R, ov_i, ov_j)
\end{aligned}
$$

❑

Beweis-Skizze zu Lemma 4.1.6, Seite 122: Fall (4.19) lässt sich direkt anhand von Lemma 4.1.4 nachweisen.

Für Forderung (4.20) formuliert kann davon ausgegangen werden, dass die beiden Objektvariablen vom selben Typ sind, da ansonsten Fall (4.19) bereits gilt. Ansonsten beschreibt (4.20) genau denn Fall in dem die beiden durch ov_j und ov_k erzeugten Objekte *mergeable* gemäß Definition 3.5.10 sind. D.h. die Objekte sind vom selben Typ und für jedes Paar erzeugter Attribute die keine Primärschlüssel sind gilt:

(i) Ein Attributwert ist $\diamondsuit$ oder

(ii) beide Werte berechnen sich innerhalb einer Regel aus gleichen Termen oder

(iii) beide Attribute haben denselben konstanten Wert.

Die Werte von Primärschlüsselattributen müssen nicht berücksichtigt werden, da unterschiedliche Werte in Primärschlüsseln sicherstellen, dass auch die zugehörigen Objekte eine unterschiedliche Identität haben. ❑

Beweis von Lemma 4.1.7, Seite 124: O.B.d.A sei $r_i, r_j \in R \in RW$, $ov_i \in r_i|_{OVB}$, $ov_j \in r_j|_{OVB}$ und $M \in \mathbb{M}_R^0$. Es gelte nun:

$$
\begin{aligned}
& PCONF : mfm_k^1|_{match_o}(ov_i)|_{oi} = mfm_l^1|_{match_o}(ov_j)|_{oi} \wedge \\
& \qquad mfm_k^1|_{match_o}(ov_i)|_{ot} = mfm_l^1|_{match_o}(ov_j)|_{ot} \wedge \\
& \qquad GL'(r_i, ov_i, k) \wedge \\
& \qquad GL'(r_j, ov_j, k) \\
& \text{ist nicht lösbar} \\
\overset{\text{Def. 3.5.17}}{\Rightarrow} & \nexists o_i, o_j \text{ mit } createsObj(o_i, ov_i, r_i, srcModel(M, r_i)) \wedge \\
& \qquad createsObj(o_j, ov_j, r_j, srcModel(M, r_j)) : \\
& \qquad o_i|_{oi} = o_j|_{oi} \wedge o_i|_{ot} = o_j|_{ot} \\
\overset{\text{Def. 4.1.10}}{\Leftrightarrow} & \neg potConflicting(R, ov_i, ov_j)
\end{aligned}
$$

❑

Beweis von Lemma 4.1.8, Seite 125: Lemma 4.1.8 besagt:

„$\forall$ Lösung für *PCONF*: Lösung erfüllt *EQATTS*“ $\Rightarrow$ „*conflictFree* gilt“

Dies lässt sich Umformen zu (Kontraposition):

„$\neg conflictFree$ gilt“ $\Rightarrow$ „$\exists$ Lösung für *PCONF*: Lösung erfüllt *EQATTS* nicht“

Diese Aussage wird im Folgenden bewiesen:

$$
\begin{aligned}
& \neg conflictFree(R, ov_i, ov_j) \\
\overset{\text{Def. 4.1.9}}{\Leftrightarrow}\quad & \exists M \in M_R^0, o_i, o_j \text{ mit } createsObj(o_i, ov_i, r_i, srcModel(M, r_i)) \wedge \\
& \qquad createsObj(o_j, ov_j, r_j, srcModel(M, r_j)) \ : \\
& \qquad\quad \neg mergeable(\{ov_i\}, \{ov_j\}) \\
\overset{\text{Def. 3.5.10}}{\Leftrightarrow}\quad & \exists M \in M_R^0, o_i, o_j \text{ mit } createsObj(o_i, ov_i, r_i, srcModel(M, r_i)) \wedge \\
& \qquad createsObj(o_j, ov_j, r_j, srcModel(M, r_j)) \ : \\
& \qquad\quad o_i|_{oi} = o_j|_{oi} \wedge o_i|_{ot} = o_j|_{ot} \wedge \\
& \qquad\quad \big(\exists (a_0, v_0) \in o_i|_V, (a_1, v_1) \in o_j|_V : \\
& \qquad\qquad a_0 = a_1 \wedge \neg attMergeable((a_0, v_0), (a_1, v_1))\big) \\
\overset{\text{Def. 4.1.12}}{\Rightarrow}\quad & \exists \text{ Lösung für } PCONF((R, ov_i, ov_j, k, l) : \\
& \qquad mfm_k^1|_{match_o}(ov_i)|_{oi} = mfm_l^1|_{match_o}(ov_j)|_{oi} \wedge \\
& \qquad mfm_k^1|_{match_o}(ov_i)|_{ot} = mfm_l^1|_{match_o}(ov_j)|_{ot} \wedge \\
& \qquad \big(\exists (a_0, v_0) \in mfm_k^1|_{match_o}(ov_i)|_V, (a_1, v_1) \in mfm_l^1|_{match_o}(ov_j)|_V : \\
& \qquad\quad a_0 = a_1 \wedge \neg attMergeable((a_0, v_0), (a_1, v_1))\big) \\
\overset{\text{Def. 3.5.9}}{\Rightarrow}\quad & \exists \text{ Lösung für } PCONF((R, ov_i, ov_j, k, l) : \\
& \qquad mfm_k^1|_{match_o}(ov_i)|_{oi} = mfm_l^1|_{match_o}(ov_j)|_{oi} \wedge \\
& \qquad mfm_k^1|_{match_o}(ov_i)|_{ot} = mfm_l^1|_{match_o}(ov_j)|_{ot} \wedge \\
& \qquad \big(\exists att \in ov_i|_{otv} : \\
& \qquad\quad mfm_k^1|_{match_o}(ov_i).att \neq mfm_l^1|_{match_o}(ov_j).att \wedge \\
& \qquad\quad \diamondsuit \notin mfm_k^1|_{match_o}(ov_i).att \cup mfm_l^1|_{match_o}(ov_j).att \\
\overset{(i)}{\Rightarrow}\quad & \exists \text{ Lösung für } PCONF((R, ov_i, ov_j, k, l) : \\
& \qquad mfm_k^1|_{match_o}(ov_i)|_{oi} = mfm_l^1|_{match_o}(ov_j)|_{oi} \wedge \\
& \qquad mfm_k^1|_{match_o}(ov_i)|_{ot} = mfm_l^1|_{match_o}(ov_j)|_{ot} \wedge \\
& \qquad \text{Lösung erfüllt } EQATTS \text{ nicht}
\end{aligned}
$$

Anmerkung zu (i)*:* Dieser Schluss ist zulässig, da nun bekannt ist, dass eine Gleichung aus *EQATTS*der Form

$$mfm_k^1|_{match_o}(ov_i).att \neq mfm_l^1|_{match_o}(ov_j).att \tag{A.6}$$

nicht erfüllt ist. In *EQATTS*sind gemäß Definition 4.1.13 keine Gleichungen enthalten, bei denen ein Attribut den Wert $\diamondsuit$ hat, was durch den Ausdruck

$$\diamondsuit \notin mfm_k^1|_{match_o}(ov_i).att \cup mfm_l^1|_{match_o}(ov_j).att$$

ausgeschlossen ist. Außerdem werden in *EQATTS*keine Primärschlüsselattribute berücksichtigt. Würde jedoch die Ungleichung (A.6) für ein Primärschlüsselattribut gelten, so wäre das Gleichungssystem *PCONF* nicht erfüllt. ❑

Beweis von Lemma 4.1.9, Seite 126: Das Lemma gilt offensichtlich wegen Definition 3.1.12 (Objektbelegung), in der festgelegt ist, dass innerhalb einer Objektbelegung jedes Objekt einen eindeutigen Identifikator haben muss, und der Definition 3.1.11 von Objekten, in der festgelegt ist, dass sich dieser Identifikator bei Objekten mit Primärschlüsselattributen aus diesen errechnet. ❑

Beweis von Lemma 4.1.10, Seite 127: Wegen $m \in \mathbb{M}_{mm_0}$ bilden die Objekte dieses Modells eine konsistente Objektbelegung.
Für jedes Paar ov_i, ov_j mit

$$ov_i, ov_j \in R'|_{mv_0}|_{OVB} \text{ mit } ov_i \neq ov_j \wedge ov_i|_{otv} = ov_j|_{otv}$$

liefern $mfm_i^0|_{match_o}(ov_i)$ und $mfm_j^0|_{match_o}(ov_j)$ jeweils Objekte aus m. Durch Einsetzen in Lemma 4.1.9 ergibt sich:

$$\begin{aligned}
& \forall att \in class|_A|_n : \\
& \quad mfm_i^0|_{match_o}(ov_i)|_{oi} = mfm_j^0|_{match_o}(ov_j)|_{oi} \\
& \quad \Rightarrow mfm_i^0|_{match_o}(ov_i).att = mfm_j^0|_{match_o}(ov_j).att \\
\overset{\text{Def. } \forall}{\Leftrightarrow} \quad & \bigwedge_{att \in class|_A|_n} \big(mfm_i^0|_{match_o}(ov_i)|_{oi} = mfm_j^0|_{match_o}(ov_j)|_{oi} \\
& \qquad \Rightarrow mfm_i^0|_{match_o}(ov_i).att = mfm_j^0|_{match_o}(ov_j).att\big) \\
\overset{\text{Implikation}}{\Leftrightarrow} \quad & \Big(\bigwedge_{att \in class|_A|_n} mfm_i^0|_{match_o}(ov_i)|_{oi} \neq mfm_j^0|_{match_o}(ov_j)|_{oi} \vee \\
& \qquad mfm_i^0|_{match_o}(ov_i).att = mfm_j^0|_{match_o}(ov_j).att\Big) \\
\Leftrightarrow \quad & mfm_i^0|_{match_o}(ov_i)|_{oi} \neq mfm_j^0|_{match_o}(ov_j)|_{oi} \vee \\
& \bigwedge_{att \in class|_A|_n} mfm_i^0|_{match_o}(ov_i).att = mfm_j^0|_{match_o}(ov_j).att
\end{aligned}$$

❑

Beweis von Satz 4.1.5, Seite 127: Offensichtlich ist Satz 4.1.5 lediglich ein Spezialfall von Lemma 4.1.8. Weiterhin ist bekannt:

(*i*) Gemäß Definition 3.5.16 sind Quellmodelle einer Modelltransformation gültige Modelle.

(*ii*) Gemäß Lemma 4.1.10 gilt Aussage $VALIDOB(R, ov_i, ov_j, k, l)$ für alle gültigen Modelle (bzw. Objektbelegungen).

Demzufolge kann mit Hilfe von Satz 4.1.5 die Konfliktfreiheit von Objektvariablen aus beliebigen Regeln mit gleichem Quellmetamodell nachgewiesen werden. ❑

Anwendbarkeit einer Regel

Beweis von Satz 4.1.6, Seite 130: Das Ergebnis einer Modelltransformation wird genau dann zu $\perp$ falls des Einfügen eines neu erzeugten Modellfragements in ein Zielmodellfragment den Wert $\perp$ liefert. Um dieses Ergebnis zu erhalten gibt es gemäß Definition 3.5.12 genau zwei Möglichkeiten:

1. Einer der beiden Parameter des $\cup_{\mathfrak{m}}$-Operators hat den Wert $\perp$.
2. Für die beiden Parameter gilt die Relation *mergeable* nicht.

entweder liefert eine Modellfragmenttransformation $\perp$ als Ergebnis oder die Merge-Operation liefert $\perp$, da unterschiedliche Attributwerte nicht gemischt werden konnten.

Fall 1: Dieser Fall kann nur dann auftreten, falls eine Modellfragmenttransformation den Wert $\perp$ als Ergebnis liefert. Da jedoch gemäß (i) für jede Regel das Prädikat *createsValidFragments* gelten muss, welches besagt, dass die Regel niemals ein Modellfragment mit dem Wert $\perp$ erzeugt (siehe Def. 4.1.3, ist dieser Fall ausgeschlossen.

Fall 2: Der Merge zweier Modellfragmente mf_0, mf_1 liefert $\perp$ wenn für diese beiden Modellfragmente gilt: $\neg mergeable(mf_0|_{OB}, mf_1|_{OB})$. Gemäß Definition 3.5.10 ist dies der Fall, falls die Objektbelegungen jeweils ein o_0 und o_1 gleichen Typs und mit gleichem Identifikator enthalten und für ein Paar ihrer Attribute *attMergeable* nicht gilt. Hierbei sind zwei Fälle zu unterschieden:

Fall 2.1: Beide Objekte wurden von derselben Objektvariablen *ov* der Regel *r* erzeugt.
In diesem Fall gilt gemäß (ii) für diese Objektvariable

$$\begin{aligned} & deterministic(r, ov) \\ \Leftrightarrow & \big(o_0|_{oi} = o_1|_{oi} \Rightarrow o_0|_V = o_1|_V\big) \\ \Rightarrow & mergeable(\{o_0\}, \{o_1\}) \end{aligned}$$

Fall 2.2: Beide Objekte wurden von verschiedenen Objektvariablen ov_i und ov_j der Regel *r* erzeugt. In diesem Fall gilt wegen (iii) für diese beiden Objektvariablen:

$$\begin{aligned} & conflictFree(\{r\}, ov_i, ov_j) \\ \Leftrightarrow & mergeable(\{o_i\}, \{o_j\}) \end{aligned}$$

Folglich können gemäß (ii) und (iii) alle erzeugten Objektbelegungen zusammengeführt ohne das Ergebnis $\perp$ zu liefern.

❑

Anwendbarkeit eines Regelwerks

Beweis-Skizze zu Satz 4.1.7, Seite 130: *Hilfsaussage:*
O.B.d.A sei $R = \{r_0, \ldots, r_n\}$

$$\begin{aligned} \text{Sei } mf &= transform(M, R) \\ &= apply(srcModel(M, r_0), r_0) \cup_{\mathfrak{m}} \cdots \cup_{\mathfrak{m}} apply(srcModel(M, r_n), r_n) \end{aligned}$$

Offensichtlich gilt $mf = \perp$ gdw. (a) eine Regelanwendung *apply* mit einem leeren Quellmodellfragment im Verlauf einer Regelwerksanwendung $\perp$ liefert, oder (b) der Merge eines neu generierten Fragments in das Zielmodellfragment $\perp$ liefert.

Fall (a): Ausgeschlossen durch (i).

Fall (b): Dies ist gemäß Definition 3.5.16, Definition 3.5.14 und Definition 3.5.12 durch (ii) ausgeschlossen.

❑

A.2.2 Metamodellkonformität

Grundlagen

Beweis-Skizze zu Lemma 4.2.1, Seite 131:
Gemäß Definition 3.1.14 existieren die folgenden möglichen Gründe dafür, dass ein Modellfragment *mf* nicht konform mit seinem jeweiligen Metamodell *mm* sein kann:

(i) *OB* passt nicht zum Metamodell *mm*

(ii) (3.15) wird verletzt, d.h. Kardinalitäten sind größer oder kleiner als die Klassenassoziationsmultiplizitäten dies zulassen.

(iii) (3.16) wird verletzt, d.h. Assoziationen haben an einem Ende ein Objekt das nicht aus *m* stammt.

(iv) *OB* ist nicht $\diamondsuit$-frei

(v) Das Ergebnis von *transform* liefert $\perp$.

Die Aussagen (i) und (iii) können nicht zutreffen, was durch die Definition eines Modellfragments und die Konstruktion von *mf* als Ergebnis einer Regelwerksanwendung sichergestellt ist. (iv) kann ebenfalls nicht zutreffen da gemäß der Definition einer Regelwerksanwendung (Def. 3.5.16) sämtliche $\diamondsuit$-Werte durch Default-Werte ersetzt werden. (v) ist durch die Tatsache, dass das Regelwerk *R* anwendbar ist ausgeschlossen. ❑

Verifikationstechniken für Obergrenzenkonformität

Beweis-Skizze zu Lemma 4.2.2, Seite 136:
$r_i|_{mv_0}$ ist eine Substruktur von $r_j|_{mv_0}$. Gemäß Definition 3.5.2 gilt, dass immer dann wenn ein Modellfragment auf $r_j|_{mv_0}$ matched, matched auch die zugehörige Substruktur des Modellfragments auf $r_i|_{mv_0}$. Das bedeutet das die korrespondierenden Objektvariablen der Modellvariablen jeweils auf dieselben Objekte matchen. ❑

Beweis-Skizze zu Lemma 4.2.3, Seite 139:
Annahme: $redundant(R, r_i, ova)$ gilt.

Gemäß Definition 4.2.11 definiert also mindestens eine Regel r_j für die gilt $r_j \sqsubseteq r_i$. Wegen Lemma 4.2.2 weiß man für ein beliebiges gegebenes Modellfragment *mf* das zu $r_i|_{mv_0}$ matched, dass die entsprechende Substruktur von *mf* auch zu $r_j|_{mv_0}$ matched.

Weiterhin existieren aufgrund von Definition 4.2.10 Paare von Objektvariablen $r_i|_{mv_1}$ und $r_j|_{mv_1}$, aus denen Objekte mit gleichen Identifikatoren erzeugt werden (Def. 4.2.10 (ii)). Da das gegebene Regelwerk anwendbar ist sind diese Objekte jeweils *mergeable*.

Folglich werden auch die von r_j erzeugten Objektassoziationen überflüssiger Weise auch von r_i erzeugt. Dabei ist es wichtig festzuhalten, dass gemäß Definition 4.2.11 die Kardinalitäten der von r_j erzeugten Assoziationen mindestens so groß wie die der von r_i erzeugten sein müssen. ❑

Beweis von Satz 4.2.1, Seite 141:

$$maxRelevant(R,r_i,ova) \Rightarrow relevant(R,r_i,ova)$$

$$\equiv$$

$$\neg\Big(maxRedundant(R,r_i,ova) \wedge \nexists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge maxRedundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\Rightarrow \neg\Big(redundant(R,r_i,ova) \wedge \nexists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge redundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\overset{\text{Kontraposition: } \neg A \Rightarrow \neg B \equiv A \Leftarrow B}{\equiv}$$

$$\Big(maxRedundant(R,r_i,ova) \wedge \nexists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge maxRedundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\Leftarrow \Big(redundant(R,r_i,ova) \wedge \nexists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge redundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\overset{\text{Implikation: } A \Rightarrow B \equiv \neg A \vee B}{\equiv}$$

$$\Big(maxRedundant(R,r_i,ova) \wedge \nexists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge maxRedundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\vee\neg\Big(redundant(R,r_i,ova) \wedge \nexists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge redundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\equiv$$

$$\Big(maxRedundant(R,r_i,ova) \wedge \nexists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge maxRedundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\vee\Big(\neg redundant(R,r_i,ova) \vee \exists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge redundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\equiv$$

$$\Big(maxRedundant(R,r_i,ova) \wedge \nexists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge maxRedundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\vee\Big(\neg redundant(R,r_i,ova) \vee \exists r_j : \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge redundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\equiv$$

$$\exists r_j :$$

$$\Big(maxRedundant(R,r_i,ova) \wedge \neg\big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge maxRedundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\vee\Big(\neg redundant(R,r_i,ova) \vee \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge redundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\equiv$$

$$\exists r_j :$$

$$\Big(maxRedundant(R,r_i,ova) \wedge \big(r_j \geq r_i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg maxRedundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\vee\neg redundant(R,r_i,ova) \vee \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge redundant(\{r_j,r_i\},r_i,ova)\big)\Big)$$

$$\equiv$$

$$\exists r_j :$$

$$\Big((maxRedundant(R,r_i,ova) \vee \neg redundant(R,r_i,ova)) \wedge \big(\neg redundant(R,r_i,ova) \vee$$

$$
\begin{aligned}
& r_j \geq r_i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg \mathit{maxRedundant}(\{r_j, r_i\}, r_i, ova))\Big) \\
& \vee \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge \mathit{redundant}(\{r_j, r_i\}, r_i, ova))\Big) \\
& \qquad \overset{\text{Lemma 4.2.3: } \mathit{redundant} \Rightarrow \mathit{maxRedundant}}{\equiv} \\
& \exists r_j : \\
& \Big(\neg \mathit{redundant}(R, r_i, ova) \vee r_j \geq r_i \vee r_j|_{mv_0} \ncong r_i|_{mv_0} \vee \neg \mathit{maxRedundant}(\{r_j, r_i\}, r_i, ova)\Big) \\
& \vee \big(r_j < r_i \wedge r_j|_{mv_0} \cong r_i|_{mv_0} \wedge \mathit{redundant}(\{r_j, r_i\}, r_i, ova))\Big) \\
& \qquad \equiv \\
& \exists r_j : \\
& \Big(\neg \mathit{redundant}(R, r_i, ova) \vee \neg \mathit{maxRedundant}(\{r_j, r_i\}, r_i, ova)\Big) \\
& \vee \big((r_j < r_i \vee j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0}) \\
& \wedge (r_j|_{mv_0} \cong r_i|_{mv_0} \vee r_j \geq r_i \vee r_j|_{mv_0} \ncong r_i|_{mv_0}) \\
& \wedge (\mathit{redundant}(\{r_j, r_i\}, r_i, ova) \vee j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0}))\Big) \\
& \qquad \equiv \\
& \exists r_j : \\
& \Big(\neg \mathit{redundant}(R, r_i, ova) \vee \neg \mathit{maxRedundant}(\{r_j, r_i\}, r_i, ova)\Big) \\
& \vee \big((\mathit{redundant}(\{r_j, r_i\}, r_i, ova) \vee j \geq i \vee r_j|_{mv_0} \ncong r_i|_{mv_0}))\Big) \\
& \qquad \equiv \\
& \exists r_j : \\
& \Big(\neg \mathit{redundant}(R, r_i, ova) \vee \mathit{redundant}(\{r_j, r_i\}, r_i, ova) \\
& \vee \neg \mathit{maxRedundant}(\{r_j, r_i\}, r_i, ova)\Big) \\
& \vee r_j \geq r_i \vee r_j|_{mv_0} \ncong r_i|_{mv_0})\Big) \\
& \qquad \equiv \\
& \exists r_j : \\
& \Big(\mathit{redundant}(R, r_i, ova) \Rightarrow \mathit{redundant}(\{r_j, r_i\}, r_i, ova) \\
& \vee \neg \mathit{maxRedundant}(\{r_j, r_i\}, r_i, ova)\Big) \\
& \vee r_j \geq r_i \vee r_j|_{mv_0} \ncong r_i|_{mv_0})\Big) \\
& \qquad \overset{\exists r_j : \mathit{redundant}(R, r_i, ova) \Rightarrow \mathit{redundant}(\{r_j, r_i\}, r_i, ova)}{\equiv} \\
& \mathit{true}
\end{aligned}
$$

❑

Beweis-Skizze zu Lemma 4.2.4, Seite 145: Falls $\mathit{cannotCreateSame}(R, \{ov_0, \ldots, ov_n\})$ gilt, trifft einer der drei Fälle aus Definition 4.2.15 zu.

Fall (i) *trifft zu:*
Dementsprechend existiert eine Objektvariable in $\{ov_0,\ldots,ov_n\}$ die in keiner Zielmodellvariable von R enthalten ist. Da eine solche Objektvariable keine Objekte erzeugen kann, können die Objektvariablen $\{ov_0,\ldots,ov_n\}$ auch nie dasselbe Objekt erzeugen.

Fall (ii) *trifft zu:*
$ov_i \not\sim ov_j$
$\overset{\text{Lemma 3.5.2}}{\Rightarrow}$ Die von ov_i und ov_j erzeugten Objekte können nicht identisch sein.

Fall (iii) *trifft zu:*
Annahme: es existiert eine eine Menge OV^0 mit den in (iii) geforderten Eigenschaften. Die Ungleichung treffe für ein beliebiges Assoziationsende $ae \in AE \in mm_0|_{CA}$ zu:

$$ub < \sum_{card\in\mathbb{N}^+}\Big(\max_{ov^0\in OV^0}\Big\{\underbrace{\sum_{\substack{ova_k\in ova_k\in R|_{mv_0}|_{OVA}:\\ ova_k|_{OVAT}=AE\ \wedge\\ (ae,ov_i^0)\in ova_k|_{OVAE}\ \wedge\\ ova_k|_{OVAE}|_{ov}\neq\{ov_i^0\}\ \wedge\\ ova_k|_{cardv}=card}} card}_{(1)}\Big\} + \max_{ov^0\in OV^0}\Big\{\underbrace{\sum_{\substack{ova_k\in ova_k\in R|_{mv_0}|_{OVA}:\\ ova_k|_{OVAT}=AE\ \wedge\\ (ae,ov_i^0)\in ova_k|_{OVAE}\ \wedge\\ ova_k|_{OVAE}|_{ov}=\{ov_i^0\}\ \wedge\\ ova_k|_{cardv}=card}} card}_{(2)}\Big\}\Big)$$

Dann bezeichnet die für jedes $ov^0 \in OV^0$ die mit (1) gekennzeichnete Menge alle *nicht* reflexiven Objektvariablenassoziation, die

- von der Objektvariablen ov^0 ausgehen,
- deren Ende vom Typ *ae* mit der Objektvariablen ov^0 verbunden ist und
- die Kardinalität *card* aufweisen.

Dementsprechend umfasst die Menge (2) alle reflexiven Objektvariablenassoziationen mit den jeweils gleichen Eigenschaften.

Aus Definition 3.5.2 (3.22) ist bekannt, dass für einen Match in einem Quellmodell Kardinalität der gefundenen Assoziationen *genau* der Kardinalität der jeweiligen Objektvariablenassoziation entsprechen muss. Wegen Definition 3.5.2 (3.21) können weiterhin keine reflexiven Assoziationen auf nicht reflexive Assoziationen matchen.

Da sämtliche Objektvariable aus OV^0 dasselbe Objekt matchen, muss dieses Objekt über zumindest genausoviele ausgehende Assoziationen des Typs *ae* verfügen wie alle von den Objektvariablen von OV^0 ausgehende Objektvariablenassoziationen. Da weiterhin Objektvariablenassoziationen unterschiedlicher Kardinalität bzw. reflexive und nicht reflexive nicht auf dieselben Objektassoziationen matchen können, müssen von dem gematchten Objekt Assoziationen jeder „Art“ die in einer der Objektvariablen auftritt ausgehen. Die „Art“ einer Assoziation ist hierbei durch ihre Kardinalität und die Tatsache ob sie reflexiv ist oder nicht bestimmt.

Die Zahl der von einem solchen Objekt ausgehenden Assoziationen wird, unter Berücksichtigung ihrer Kardinalität, in der Ungleichung von Punkt (iii) aufsummiert. Ist dieser Wert größer als die jeweilige Multiplizitätsobergrenze, so bedeutet diese, dass kein Objekt eins zu mm_0 konformen Modells je von *allen* Objektvariablen aus OV^0 gematcht werden kann. Aufgrund der Implikation in (iii) können dann die Objektvariablen $\{ov_0,\ldots,ov_n\}$ nie alle dasselbe Objekt im Zielmodell erzeugen. ❑

Beweis von Lemma 4.2.5, Seite 146: Wegen Definition 4.2.14 gilt:

$$\begin{aligned}\mathbb{OV}_R^{conf*} &= \{OV \in \mathbb{OV}_R : maxCreateSameObj(R,OV)\}\\ &\subseteq \{OV \in \mathbb{OV}_R : createSameObj(OV)\}\end{aligned} \tag{A.7}$$

Gemäß Lemma 4.2.4 gilt (Kontraposition):

$$
\begin{aligned}
& createSameObj(R,OV) \Rightarrow \neg cannotCreateSame(R,OV) \\
\Rightarrow\ & \forall OV' \in \{OV \in \mathbb{OV}_R : createSameObj(R,OV)\} : \\
& \quad \exists OV^* \in \underbrace{\{OV \in \mathbb{OV}_R : \neg cannotCreateSame(R,OV)\}}_{=\mathbb{OV}_R^{conf*}} : OV' \subseteq OV^* \\
\overset{(A.7)}{\Rightarrow}\ & \forall OV'' \in \overbrace{\{OV \in \mathbb{OV}_R : maxCreateSameObj(R,OV)\}}^{=\mathbb{OV}_R^{conf}} : \\
& \quad \exists OV^* \in \mathbb{OV}_R^{conf*} : OV'' \subseteq OV^* \\
\Leftrightarrow\ & \forall OV \in \mathbb{OV}_R^{conf} : \\
& \quad \exists OV^* \in \mathbb{OV}_R^{conf*} : OV \subseteq OV^*
\end{aligned}
$$

❑

Beweis von Satz 4.2.2, Seite 152:

$$
\begin{aligned}
& \forall AE, ae \text{ mit } AE \in mm_R^1|_{CA}, ae \in AE : maxCard(AE,ae,R) \leq ub, \\
& \text{mit } ae|_m = (lb,ub) \\
\overset{\text{Def. 4.2.19}}{\Rightarrow}\ & \max_{M \in \mathbb{M}_R^0} (\max_{o \in transform(M,R)|_{OB}} (\sum_{oa:(oppositeEnd(AE,ae),o) \in oa|_{OAE}} oa|_{card})) \leq ub \\
& \text{mit } ae|_m = (lb,ub) \\
\overset{\text{"}\max_i \max_j \sum_{k(i,j)} card \geq \forall_i \forall_j \sum_{k(i,j)} card\text{"}}{\Rightarrow}\ & \forall AE \in mm_R^1|_{CA} : \forall ae \in AE : \forall M \in \mathbb{M}_R^0 : \\
& \forall o \in transform(M,R)|_{OB} \text{ mit } ae|_c \in types(o|_{ot}) : \\
& \sum_{oa:(oppositeEnd(AE,ae),o) \in oa|_{OAE}} oa|_{card} \leq ub \text{ mit } ae|_m = (lb,ub) \\
\overset{\text{Def. 4.2.2}}{\Rightarrow}\ & \forall M \in \mathbb{M}_R^0 : ubConform(transform(M,R),mm) \\
\overset{\text{Def. 4.2.6}}{\Rightarrow}\ & ubConform(R)
\end{aligned}
$$

❑

Beweis von Lemma 4.2.6, Seite 152: *Hilfsbehauptung 1:* Generell gilt:

$$
\max_k \left(\sum_l f(k,l) \right) \leq \sum_l \max_k f(k,l) \tag{A.8}
$$

Hilfsbehauptung 2: Sei $M \in \mathbb{M}_R^0$ und $o \in transform(M,R)$ ein beliebiges Objekt. Weiter sei ov_0 eine

beliebige Objektvariable die o erzeugt haben könnte.

$$\begin{aligned}
\exists OV_k \in \mathbb{OV}_R^{conf} : \Big\{ & oa : oa \in \bigcup_{\substack{\mathrm{m} \\ r_i : r_i \in R}} apply(srcModel(M, r_i), r_i) \\
& \wedge (oppositeEnd(AE, ae_1), o) \in oa|_{OAE} \Big\} \\
\subseteq \\
\Big\{ & oa : (\exists r_i \in R : \exists mfm_\mu^1(ova) = oa \\
& \wedge (oppositeEnd(ova|_{OVAT}, ae_1), o) \in o) \in oa|_{OAE} \\
& \wedge ae_1 \in ova|_{OVAT}|_{ae_1} \\
& \wedge ova \in \{ova' : \exists ov_0' \in r_i|_{OVB} \wedge maxRelevant(R, r_i, AE, ae_1, ova', ov_0, ov_0') \wedge \\
& \qquad \exists (ov_0, r_i) \in OV_k \wedge mfm_\mu^1(ov_0) = o\} \Big\}
\end{aligned} \tag{A.9}$$

Aussage (A.9) besagt, dass jede Objektassoziation in der ersten Menge auch Teil der zweiten Menge ist. Dies ist zum einen eine direkte Konsequenz aus der Definition von *maxRedundant* (Def. 4.2.7), der Definition von $\mathbb{OV}_R^{conf}$ (Def. 4.2.16) und der Definition von *merge* (Def. 3.5.12). *maxRedundant* identifiziert sämtliche redundante Objektvariablenassoziationen. Diese redundanten Objektvariablenassoziationen sind genau jene, welche von OA_2 gemäß der Definition von *merge* weggenommen werden (siehe Def. 3.5.12, S. 95). Zu jedem erzeugten Objekt existiert weiterhin eine Menge OV_k mit den Objektvariablen, die es erzeugt haben können. Alle anderen Objektvariablen können dann gemäß der Definition von $\mathbb{OV}_R^{conf}$ das Objekt mit Sicherheit nicht erzeugt haben. Dementsprechend brauchen auch die Objektvariablenassoziationen an diesen Objektvariablen nicht weiter in Betracht gezogen werden.

$$\begin{aligned}
& \max_{\substack{OV_k \in \mathbb{OV}_R^{conf} : \forall ov_m \in OV_k : \\ oppositeEnd(AE_i, ae_j)|_c \in types(ov_m|_{otv})}} \Big\{ \sum_{\substack{r_l \in R, ova_m \in r_l|_{mv_1}|_{OVA} : \exists ov_h \in OV_k : \\ \big(ov_h \in r_l|_{mv_1}|_{OVB} \\ \quad \wedge (oppositeEnd(AE_i, ae_j), ov_h) \in ova|_{OVAE} \\ \quad \wedge ova_m|_{OVAT} = AE_i \\ \quad \wedge maxRelevant(R, r_l, ova_m)\big)}} maxVarCard(ova_m, ae_j, r_l) \Big\} \\
\overset{\text{Def. 4.2.20}}{=} & \max_{\substack{OV_k \in \mathbb{OV}_R^{conf} : \forall ov_m \in OV_k : \\ oppositeEnd(AE_i, ae_j)|_c \in types(ov_m|_{otv})}} \\
& \Big\{ \sum_{\substack{r_l \in R, ova_m \in r_l|_{mv_1}|_{OVA} : \exists ov_h \in OV_k : \\ \big(ov_h \in r_l|_{mv_1}|_{OVB} \\ \quad \wedge (oppositeEnd(AE_i, ae_j), ov_h) \in ova|_{OVAE} \\ \quad \wedge ova_m|_{OVAT} = AE_i \\ \quad \wedge maxRelevant(R, r_l, ova_m)\big)}} \max_{M \in \mathbb{M}_R^0} \Big(\max_{o \in apply(srcModel(M, r_l), r_l)|_{OB}} (numAssos(r_l, m, o, ova_m, ae_1)) \Big) \Big\} \\
\overset{\sum_{a,b} = \sum_a \sum_b}{=} & \max_{\substack{OV_k \in \mathbb{OV}_R^{conf} : \forall ov_m \in OV_k : \\ oppositeEnd(AE_i, ae_j)|_c \in types(ov_m|_{otv})}} \Big\{ \sum_{r_l \in R} \sum_{\substack{ova_m \in r_l|_{mv_1}|_{OVA} : \exists ov_h \in OV_k : \\ \big(ov_h \in r_l|_{mv_1}|_{OVB} \\ \quad \wedge (oppositeEnd(AE_i, ae_j), ov_h) \in ova|_{OVAE} \\ \quad \wedge ova_m|_{OVAT} = AE_i \\ \quad \wedge maxRelevant(R, r_l, ova_m)\big)}}
\end{aligned}$$

$$\max_{M\in\mathbb{M}_R^0}\Big(\max_{o\in apply(srcModel(M,r_l),r_l)|_{OB}} numAssos(r_l,m,o,ova_m,ae_1))\Big\}$$

$$\overset{\text{Def. 4.2.5}}{=} \max_{\substack{OV_k\in\mathbb{OV}_R^{conf}:\forall ov_m\in OV_k:\\ oppositeEnd(AE_i,ae_j)|_c\in types(ov_m|_{otv})}} \Big\{ \sum_{r_l\in R} \sum_{\substack{ova_m\in r_l|_{mv_1}|_{OVA}:\exists ov_h\in OV_k:\\ \big(ov_h\in r_l|_{mv_1}|_{OVB}\\ \wedge(oppositeEnd(AE_i,ae_j),ov_h)\in ova|_{OVAE}\\ \wedge ova_m|_{OVAT}=AE_i\\ \wedge maxRelevant(R,r_l,ova_m)\big)}}$$

$$\max_{M\in\mathbb{M}_R^0}\Big(\max_{o\in apply(srcModel(M,r_l),r_l)|_{OB}} ova_m|_{cardv}\cdot\big|\{oa:\exists mfm_\mu^1\in MFM^1(m,r_l)$$
$$\text{mit } mfm_\mu^1|_{match_a}(ova_m)=oa$$
$$\wedge(oppositeEnd(ova_m|_{OVAT},ae_1),o)\in oa|_{OAE}$$
$$\wedge ae_1\in ova_m|_{OVAT}|_{ae}\}\big|\quad\Big)\Big\}$$

$$\overset{\text{(A.8), (A.9)}}{\geq} \max_{M\in\mathbb{M}_R^0}\Big(\max_{o\in\bigcup_{r_l\in R} apply(srcModel(M,r_l),r_l)|_{OB}}\Big(\sum_{oa:(oppositeEnd(AE,ae_1),o)\in oa|_{OAE}} oa|_{card}\Big)\Big)$$

$$\overset{\text{Def. 3.5.16}}{=} \max_{M\in\mathbb{M}_R^0}\Big(\max_{o\in transform(M,R)|_{OB}}\Big(\sum_{oa:(oppositeEnd(AE,ae_1),o)\in oa|_{OAE}} oa|_{card}\Big)\Big)$$

$$\overset{\text{Def. 4.2.19}}{=} maxCard(AE,ae_1,R)$$

❑

Beweis von Lemma 4.2.7, Seite 155: Sei ae_0, ov_0, ov_1 so gewählt wie in Definition 4.2.21.

1. Fall: $ov_0 = ov_1$

- (i) $dependsOn(ov_0,r_i)=\emptyset$, d.h. $ov_0|_{oiv}$ ist fest: Der Beweis ergibt sich direkt aus Lemma 3.5.10 (i).
- (ii) $dependsOn(ov_0,r_i)=\perp$, d.h. $ov_0|_{oiv}$ ist frei: Der Beweis ergibt sich direkt aus Lemma 3.5.10 (i).
- (iii) $dependsOn(ov_0,r_i)=\{ov_i',...,ov_j'\}\neq\emptyset$, d.h. der Wert des Objektvariablenidentifikatorterms $ov_0|_{oiv}$ ist eineindeutig abhängig von $\{ov_i',...,ov_j'\}$: Der Beweis ergibt sich direkt aus Lemma 3.5.10 (i).
- (iv) $dependsOn(ov_0,r_i)=\diamondsuit$ gilt trivialerweise wegen der Definition von $\diamondsuit$ und Lemma 3.5.10 (i).

2. Fall: $ov_0 \neq ov_1$

Sei $m\in\mathbb{M}_{r_i|_{mv_0}|_{mm}}$ beliebig.

(i-iii, x)

$$dependsOn(ov_1,r_i)=\emptyset \overset{\text{Def. } dependsOn}{\Rightarrow} ov_1|_{oiv}=konst. \tag{a}$$
$$\overset{\text{Def. } mft,\ merge}{\Rightarrow} \exists_1 o'\in apply(m,r_i)|_{OB}: mfm_\mu^1|_{match_o}(ov_1)=o'$$

d.h. es wird genau ein Objekt o' von ov_1 erzeugt.

$$\forall oa \in apply(m,r_i)|_{OA} : (ae_1, mfm^1_\mu|_{match_o}(ov_1)) \in oa|_{AE} \wedge mfm^1_\mu|_{match_a}(ova) = oa \quad \text{(b)}$$

d.h. alle Assoziationen die von einer Objektvariablenassoziation erzeugt wurden haben $(ae_1, mfm^1_\mu|_{match_o}(ov_1))$ als ein Ende. Daher enthält ein Ende das Objekt o' das von der Objektvariablen ov_1 erzeugt wurde.

$$\begin{aligned} &\overset{\text{(a) und (b)}}{\Rightarrow} (ae_1, o') \text{ ist ein Ende aller aller aus } ova \text{ erzeugten Assoziationen} \\ &\Rightarrow \quad numAssos(r_i, m, o, ova, ae_1) \overset{\text{Def. 4.2.5}}{=} \\ &\qquad = ova|_{cardv} \cdot \left|\{((ae_1,o'),(ae^0,o)),...,((ae_1,o'),(ae^n,o))\}\right| \end{aligned}$$

Dabei gilt $o' \neq o$, da

$$o = mfm^1_\mu|_{match_o}(ov_0) \wedge o' = mfm^1_\mu|_{match_o}(ov_1) \wedge ov_0 \neq ov_1$$

gemäß der Annahme für den 2. Fall.
Es muss gelten:

$$\begin{aligned} &\forall oa \in \{(ae_1,o'),(ae^i,o^i)\} : \\ &\quad oa|_{AE} = \{(ae_1,o'),(oppositeEnd(ova|_{OVAT}, ae_1), o)\} \text{ (Def. 4.2.5)} \\ &\Rightarrow \text{Es existiert maximal ein Element } ((ae_1,o'),(ae^i,o^i)) \end{aligned}$$

$$\begin{aligned} &\forall o : o \in apply(m,r_i)|_{OB} \wedge mfm^1_\mu|_{match_o}(ov_0) = o \text{ es gilt:} \\ &\quad numAssos(r_i,m,o,ova,r_i) \leq ova|_{cardv} \\ &\Rightarrow maxVarCard(ova,ae,r_i) \leq \max\{\max\{0, ova|_{cardv}\}\} \\ &\qquad\qquad = ova|_{cardv} \\ &\qquad\qquad = ubVarCard(ova, ae_1, r_i) \end{aligned}$$

(iv,v,vii,viii, xiv,xv) trivial, wegen $\infty \geq maxVarCard(ova, ae_1, r_i)$.

(vi, xvi) $dependsOn(ov_0, r_i) = M_0 \neq \emptyset$

Substitution/Umstellen der Definition von *numAssos* liefert

$$\begin{aligned} &numAssos(r_i,m,o,ova,ae_1) \\ &\quad = ova|_{cardv} \cdot \left|\{\{(ae_0,o),(ae_1,o^0)\},...,\{(ae_0,o),(ae_1,o^n)\}\}\right| \\ &\quad \leq ova|_{cardv} \cdot (n+1) \quad \text{(c)} \\ &\text{mit } mfm^1_\mu|_{match_a}(ova) = oa = \{(ae_0,o),(ae_1,o^i)\} \\ &\quad \text{wobei } (ae_0,o) = oppositeEnd(ova|_{OVAT}, ae_1) \end{aligned}$$

$$\begin{aligned} &\forall oa_i = \{(ae_0,o),(ae_1,o^i)\} : \exists mfm^1_\mu \in MFM^1 : \\ &\quad mfm^1_\mu|_{match_a}(ova) = oa_i \wedge mfm^1_\mu|_{match_o}(ov_0) = o \quad \text{(d)} \\ &\quad \wedge\ mfm^1_\mu|_{match_o}(ov_1) = o^i \end{aligned}$$

Für jede Objektassoziation oa existiert ein mfm^1_μ, der oa "erzeugt" hat. Der zugehörige Assoziations-Match $match_{\mathrm{a}}(ova) = oa_i$ bildet immer auf oa_i ab, während der Objekt-Match $match_{\mathrm{o}}(ov_1) = o^i$ für verschiedene Assoziationen oa_j, oa_i auch auf dasselbe Objekt o' abbilden kann, d.h. $mfm^1_{\mu_i}|_{match_{\mathrm{o}}}(ov_1) = o' = mfm^1_{\mu_j}|_{match_{\mathrm{o}}}(ov_1)$

Die Menge $\{\{(ae_0,o),(ae_1,o^0)\},...,\{(ae_0,o),(ae_1,o^n)\}\}$ hat maximal soviel Elemente wie verschiedene o^i's existieren können. D.h. es gilt für $numAssos(r_i,m,o,ova,ae_1)$ aus (c):

$$numAssos(r_i,m,o,ova,ae_1) = (n+1)\cdot ova|_{cardv}, o^i \neq o^j \forall 0 \leq i,j \leq n$$

Aus (d) folgt:

$$\begin{aligned}
&\left|\{oa_i : oa_i = \{(ae_0,o),(ae_1,o^i)\}\}\right| \overset{(d)}{\leq} n+1 \overset{\text{Def. 3.5.14}}{=} |MFM^1| \\
&dependsOn(ov_0,r_i) = M_0 \\
&\Rightarrow n+1 \leq maxmatch(r_i|_{mv_0}, M_0, \emptyset) \\
&\Rightarrow \max_{o\in apply(m,r_i)|_{OB}} (numAssos(r_i,m,o,ova,ae_1)) = ova|_{cardv}\cdot(n+1) \\
&\qquad\qquad \leq ova|_{cardv}\cdot maxmatch(r_i|_{mv_0}, M_0, \emptyset)
\end{aligned}$$

Da m fest aber beliebig ist gilt auch:

$$\begin{aligned}
&\max_{m\in\mathbb{M}_{r_i|_{mv_0}|_{mm}}} \Big(\max_{o\in apply(m,r_i)|_{OB}} (numAssos(r_i,m,o,ova,ae_1))\Big) \\
&\qquad\qquad \leq ova|_{cardv}\cdot maxmatch(r_i|_{mv_0}, M_0, \emptyset)
\end{aligned}$$

(ix) **1. Fall:** $M_0 \overset{r_i|_{mv_0}}{\rightsquigarrow} M_1$

Sei o' ein Objekt das aus der Objektvariable ov_0 erzeugt wurde. Wegen

$$dependsOn(ov_0,r_i) = M_0$$

weiß man, dass immer wenn die Elemente von M_0 auf dieselben Quellobjekte matchen dasselbe Zielobjekt o' erzeugt wird. Wegen

$$dependsOn(ov_1,r_i) = M_1$$

weiß man, dass immer wenn die Elemente von M_1 auf dieselben Quellobjekte matchen, dasselbe Zielobjekt o'' generiert wird. Weiterhin legt $M_0 \overset{r_i|_{mv_0}}{\rightsquigarrow} M_1$ fest, dass immer wenn die Elemente von M_0 auf dieselben Quellobjekte matchen, matchen auch alle Elemente von M_1 eineindeutig auf dieselben Quellobjekte. Somit steht fest, dass immer wenn o' erzeugt wird, auch genau ein zugehöriges Objekt o'' erzeugt wird, welches über eine Objektassoziation des Typs von ova mit o'' verbunden ist.

2. Fall: Der Beweis verläuft analog zu den Fällen *(vi, xvi)* aber es gilt nun zusätzlich $dependsOnId(ov_1,r_i) = M_1$.
Jedesmal wenn die Objektvariablen in M_1 dieselben Objekte matchen wird dasselbe Objekt aus ov_1 erzeugt.

$\Rightarrow$ Es wird mehrfach dieselbe Assoziation $\{(ae_0,o),(ae_1,o')\}$ erzeugt.
$\Rightarrow$ $maxmatch(r_i|_{mv_0},M_0,M_1$ ignoriert die Fälle in denen M_1 auf dieselben Objekte matcht.
$\Rightarrow$ Behauptung

(xi, xii, xiii)

$$
\begin{aligned}
&dependsOn(ov_0,r_i)=\Diamond\\
&\Rightarrow \forall \mu,\tau:\mu\neq\tau \Rightarrow mfm^1_\mu|_{match_o}(ov_0)\neq mfm^1_\tau|_{match_o}(ov_0)\\
&\Rightarrow \text{ für beliebige aber feste } o,\mu \text{ mit } mfm^1_\mu|_{match_o}(ov_0)=o \text{ gilt:}\\
&\qquad \exists_1 mfm^1_\mu|_{match_a}(ova)=oa \text{ mit } (ae_1,o)\in oa|_{OAE}\\
&\Rightarrow ubVarCard(ova,ae_1,r_i)=ova|_{cardv}
\end{aligned}
$$

❑

Beweis von Satz 4.2.3, Seite 155: $\forall AE,ae$ mit $AE\in mm^1_R|_{CA}, ae\in AE und ae|_m=(lb,ub)$:

$$
\begin{aligned}
&ub\\
&\overset{\text{Annahme}}{\geq} \max_{\substack{OV_k\in\mathbb{OV}_R^{conf*}:\forall ov_m\in OV_k:\\ oppositeEnd(AE_i,ae_j)|_c\in types(ov_m|_{otv})}} \Big\{ \sum_{\substack{r_l\in R, ova_m\in r_l|_{mv_1}|_{OVA}:\exists ov_h\in OV_k:\\ \left(ov_h\in r_l|_{mv_1}|_{OVB}\right.\\ \wedge(oppositeEnd(AE_i,ae_j),ov_h)\in ova|_{OVAE}\\ \wedge ova_m|_{OVAT}=AE_i\\ \left.\wedge relevant(R,r_l,ova_m)\right)}} ubVarCard(ova_m,ae_j,r_l)\Big\}\\
&\overset{\text{Lemma 4.2.5}}{\geq} \max_{\substack{OV_k\in\mathbb{OV}_R^{conf}:\forall ov_m\in OV_k:\\ oppositeEnd(AE_i,ae_j)|_c\in types(ov_m|_{otv})}} \Big\{ \sum_{\substack{r_l\in R, ova_m\in r_l|_{mv_1}|_{OVA}:\exists ov_h\in OV_k:\\ \left(ov_h\in r_l|_{mv_1}|_{OVB}\right.\\ \wedge(oppositeEnd(AE_i,ae_j),ov_h)\in ova|_{OVAE}\\ \wedge ova_m|_{OVAT}=AE_i\\ \left.\wedge relevant(R,r_l,ova_m)\right)}} ubVarCard(ova_m,ae_j,r_l)\Big\}\\
&\overset{\text{Satz 4.2.1}}{\geq} \max_{\substack{OV_k\in\mathbb{OV}_R^{conf}:\forall ov_m\in OV_k:\\ oppositeEnd(AE_i,ae_j)|_c\in types(ov_m|_{otv})}} \Big\{ \sum_{\substack{r_l\in R, ova_m\in r_l|_{mv_1}|_{OVA}:\exists ov_h\in OV_k:\\ \left(ov_h\in r_l|_{mv_1}|_{OVB}\right.\\ \wedge(oppositeEnd(AE_i,ae_j),ov_h)\in ova|_{OVAE}\\ \wedge ova_m|_{OVAT}=AE_i\\ \left.\wedge maxRelevant(R,r_l,ova_m)\right)}} ubVarCard(ova_m,ae_j,r_l)\Big\}\\
&\overset{\text{Satz 4.2.7}}{\geq} \max_{\substack{OV_k\in\mathbb{OV}_R^{conf}:\forall ov_m\in OV_k:\\ oppositeEnd(AE_i,ae_j)|_c\in types(ov_m|_{otv})}} \Big\{ \sum_{\substack{r_l\in R, ova_m\in r_l|_{mv_1}|_{OVA}:\exists ov_h\in OV_k:\\ \left(ov_h\in r_l|_{mv_1}|_{OVB}\right.\\ \wedge(oppositeEnd(AE_i,ae_j),ov_h)\in ova|_{OVAE}\\ \wedge ova_m|_{OVAT}=AE_i\\ \left.\wedge maxRelevant(R,r_l,ova_m)\right)}} maxVarCard(ova_m,ae_j,r_l)\Big\}\\
&\overset{\text{Lemma 4.2.6}}{\geq} maxCard(AE,ae,R)\\
&\overset{\text{Satz 4.2.2}}{\Rightarrow} ubConform(R)
\end{aligned}
$$

❑

Verifikationstechniken für Untergrenzenkonformität

Beweis-Skizze zu Lemma 4.2.8, Seite 156: Der Beweis stützt sich auf die Eigenschaften (i) und (ii):

(i) wegen Definition 3.5.4 (3.24) und Definition 3.5.2 (3.21).

(ii) wegen Definition 3.5.2 (3.22) und Definition 3.5.2 (3.21). ❑

Beweis von Lemma 4.2.9, Seite 157: Sei

$$\{mfm_1, \ldots, mfm_n\} := MFM(m, r_i|_{mv_0})$$

Dann gilt (Def. 3.5.14):

$$apply(m, r_i) = \bigcup_{\substack{\mathfrak{m} \\ mfm_j \in \{mfm_0, \ldots, mfm_n\}}} mft(mfm_j, r_i)$$

Weiter sei

$$mf_k := \bigcup_{\substack{\mathfrak{m} \\ mfm_j \in \{mfm_0, \ldots, mfm_k\}}} mft(mfm_j, r_i)$$

Beweis durch Induktion über die generierten Modellfragmente mf_j

Induktionsvoraussetzung, gemäß Definition 3.5.14:

$$mf_0 = mft(mfm_0, r_i)$$

Gemäß Definition 3.5.5 existieren drei mögliche Fälle:

1. Fall: $mf_0 = \perp$*:*

Ausgeschlossen, da r_i anwendbar ist.

2. Fall: $mf_0 = \emptyset$*:*

Es wurde kein Objekt o erzeugt. Da Aussagen über die leere Menge immer wahr sind, gilt die Aussage.

3. Fall: $\perp \neq mf_0 \neq \emptyset$*:*

Wegen Definition 3.5.5 muss eine Modellfragmentrelation existieren mit $mfr(mfm_0, r_i, mf_0)$. Gemäß Definition 3.5.4 muss für diese Modellfragmentrelation weiterhin ein Ziel-Match

$$mfm_v^1 \in MFM(mf_0, r_i|_{mv_1})$$

existieren, der jeder Objektvariablenassoziation aus $r_i|_{mv_1}|_{OVA}$ bijektiv eine erzeugte Objektassoziation aus $mf_0|_{OA}$ zuordnet. D.h. für beliebige aber feste o, ov' mit

$$createsObj(o, ov', r_i, m)$$

gilt gemäß Lemma 4.2.8 für beliebige aber feste AE und ae:

$$\sum_{\substack{ova \in r_i|_{mv_1}|_{OVA}: \\ ova|_{OVAT} = AE \wedge \\ (ae, ov') \in ova|_{OVAE}}} ova|_{cardv} = \sum_{\substack{oa \in mf_0|_{OA}|_{oae}: \\ oa|_{OAT} = AE \wedge \\ (ae, o) \in oa|_{OAE}}} oa|_{card}$$

Falls ov das Objekt o erzeugt gilt sicherlich $ov \in \overline{OV}$, da $\overline{OV}$ *alle* Zielobjektvariablen der Regel r_i vom gleichen Typ wie o enthält. Also gilt ($a = b \Rightarrow min(\{a\} \cup A) \leq b$):

$$\min_{ov \in \overline{OV}} \Big(\sum_{\substack{ova \in r_i|_{mv_1}|_{OVA}: \\ ova|_{OVAT} = AE \wedge \\ (ae,ov) \in ova|_{OVAE}}} ova|_{cardv} \Big) \leq \sum_{\substack{oa \in mf_0|_{OA}|_{oae}: \\ oa|_{OAT} = AE \wedge \\ (ae,o) \in oa|_{OAE}}} oa|_{card}$$

Da o, AE und ae beliebig aber fest gewählt wurden, gilt die Aussage für mf_0

Induktionsschritt:

$$mf_j := (mf_{,j-1} \cup_{\mathfrak{m}} \underbrace{mft(mfm_j, r_i)}_{=mf})$$

Die Aussage gilt für mf_{j-1}.

$$\min_{ov \in \overline{OV}} \Big(\sum_{\substack{ova \in r_i|_{mv_1}|_{OVA}: \\ ova|_{OVAT} = AE \wedge \\ (ae,ov) \in ova|_{OVAE}}} ova|_{cardv} \Big) \leq \sum_{\substack{oa \in mf_{j-1}|_{OA}|_{oae}: \\ oa|_{OAT} = AE \wedge \\ (ae,o) \in oa|_{OAE}}} oa|_{card}$$

$$\overset{\text{Lemma 3.5.10 } (i)}{\leq} \sum_{\substack{oa \in mf_j|_{OA}|_{oae}: \\ oa|_{OAT} = AE \wedge \\ (ae,o) \in oa|_{OAE}}} oa|_{card}$$

❑

Beweis von Satz 4.2.4, Seite 159: Sei $varLbConform(r_i|_{mv_1})$, $m \in \mathbb{M}|_{r_i}|_{mv_0}|_{mm}$ beliebig aber fest und $mf := apply(m, r_i)$

$\forall o \in mf|_{OB}, AE \in r_i|_{mm}|_{CA}, ae \in AE$ mit $ae|_c \in types(o|_t)$:

$\overline{OV} := \{ov \in r_i|_{mv_1}|_{OVB} : ov|_{otv} = o|_{ot}\}$

$$\sum_{\substack{oa \in mf|_{OA}: \\ oa|_{OAT} = AE \wedge \\ (ae,o) \in oa|_{OAE}}} oa|_{card} \overset{\text{Lemma 4.2.9}}{\geq} \min_{ov \in \overline{OV}} \Big(\underbrace{\sum_{\substack{ova \in r_i|_{mv_1}|_{OVA}: \\ ova|_{OVAT} = AE \wedge \\ (ae,ov) \in ova|_{OVAE}}} ova|_{cardv}}_{\geq lb \text{ wg. } varLbConform(r_i|_{mv_1})} \Big) \overset{\forall a \in A: a \geq lb \Rightarrow min(A) \geq lb}{\geq} lb$$

$$\overset{\text{Def. 4.2.3}}{\Rightarrow} lbConform(apply(m, r_i), mm_1)$$

$$\overset{\text{Def. 4.2.23}}{\Leftrightarrow} lbConform(r_i)$$

❑

Beweis-Skizze zu Lemma 4.2.10, Seite 160: Lemma 4.2.10 ist direkt aus Definition 4.2.25 ersichtlich. ❑

Beweis-Skizze zu Lemma 4.2.11, Seite 161: Allgemein gilt:

$$\min_k \Big(\sum_l f(k,l) \Big) \geq \min_{k,l} \Big(f(k,l) \Big)$$

Einsetzen in die Definition liefert die gewünschte Aussage. ❑

Beweis von Lemma 4.2.13, Seite 164:

$$\begin{aligned} minCard(AE, ae_1, r_i) &\overset{\text{Lemma 4.2.11}}{\geq} \min_{ova:ova|_{OVAT}=AE} minVarCard(ova, ae_1, r_i) \\ &\overset{\text{Lemma 4.2.12}}{\geq} \min_{ova:ova|_{OVAT}=AE} lbVarCard(ova, ae_1, r_i) \\ &\overset{\text{Def. 4.2.29}}{=} lbCard(AE, ae_1, r_i) \end{aligned}$$

❑

Beweis von Satz 4.2.5, Seite 165: Sei $m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}$ beliebig und $mf = apply(m, r_i)$. Es gilt:

$$\begin{aligned} &\forall o \in mf|_{OB}: \\ &\sum_{\substack{oa \in mf|_{OA}: \\ oa|_{oat}=AE \\ \wedge (ae_1,o) \in oa|_{OAE}}} oa|_{card} \overset{\text{Lemma 4.2.10}}{\geq} minCard(AE, ae, r_i) \overset{\text{Lemma 4.2.13}}{\geq} lbCard(AE, ae, r_i) \geq lb \\ &\overset{\text{Def. 4.2.3}}{\Rightarrow} lbConform(r_i) \end{aligned}$$

❑

Beweis von Satz 4.2.6, Seite 165:

$$\begin{aligned} &\forall r_i \in R : lbConform(r_i) \\ &\overset{\text{Def. 4.2.23}}{\Rightarrow} \forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}}, r_i \in R : lbConform(apply(m, r_i), r_i|_{mv_0}|_{mm}) \\ &\overset{\text{Lemma 3.5.10}}{\Rightarrow} \forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}} : lbConform(\bigcup_{\mathfrak{m}, r_i \in R} apply(m, r_i)) \\ &\overset{\text{Satz 3.5.16}}{\Rightarrow} \forall m \in \mathbb{M}_{r_i|_{mv_0}|_{mm}} : lbConform(transform(m, R)) \\ &\overset{\text{Def. 4.2.22}}{\Rightarrow} lbConform(R) \end{aligned}$$

❑

Beweis von Lemma 4.2.14, Seite 166: Da $R_0 \cup R_1$ anwendbar ist, muss weiterhin gelten:

$$\begin{aligned} &R_0 \cup R_1 \in \mathbb{R} \\ &\overset{\text{Def. 3.3.10 (3.18)}}{\Rightarrow} mm^1_{R_0} = mm^1_{R_1} \end{aligned} \tag{A.10}$$

Weiterhin gilt:

$$
\begin{aligned}
& lbConform(R_0) \wedge lbConform(R_1) \\
\overset{\text{Def. 4.2.22}}{\Rightarrow} \quad & \forall M \in \mathbb{M}^0_{R_0 \cup R_1} : \\
& \quad (lbConform(transform(M, R_0), mm^1_{R_0}) \wedge \\
& \quad lbConform(transform(M, R_0), mm^1_{R_1}) \quad) \\
\overset{\substack{\text{Lemma 3.5.10 } (i), \text{ (A.10)} \\ R_0 \cup R_1 \text{ anwendbar}}}{\Rightarrow} \quad & \forall M \in \mathbb{M}^0_{R_0 \cup R_1} : \\
& \quad lbConform(transform(M, R_0) \cup_{\mathfrak{m}} transform(M, R_1), mm^1_{R_0}) \\
\overset{\text{Satz 3.5.16}}{\Rightarrow} \quad & \forall M \in \mathbb{M}^0_{R_0 \cup R_1} : \\
& \quad lbConform(transform(M, R_0 \cup R_1), mm^1_{R_0}) \\
\overset{\text{Def. 4.2.22}}{\Rightarrow} \quad & lbConform(R_0 \cup R_1)
\end{aligned}
$$

❑

Verifikationstechnik für Metamodellkonformität

Beweis-Skizze zu Satz 4.2.7, Seite 166: Gemäß Lemma 4.2.1 müssen die Kardinalitäten eines Modellfragments zu den im Metamodell definierte Multiplizitäten passen um sicherzustellen, dass ein neu generiertes Modellfragment ein Modell ist.

Satz 4.2.7 ist eine direkte Folge aus der Definition von $lbConform(R)$ (s. Def. 4.2.3) und der Definition von $ubConform(R)$ (s. Def. 4.2.2). ❑

A.2.3 Bijektivität

Beweis von Satz 4.3.1, Seite 169: Das Regelwerk $R \in \mathbb{RW}$ streng match-bijektiv bezüglich des Quellmetamodells mm_0. O.B.d.A sei $M \in \mathbb{M}^0_R$ eine beliebige aber feste Menge von Quellmodellen für R. $R' = \{r_i \in R : r_i|_{mv_0} = mm_0\}$

$$
\begin{aligned}
& transform\big(transform(M, R), R^{-1}_{mm_0}\big) \\
\overset{\text{Def. 3.5.14, 3.5.16}}{=} \quad & \bigcup_{\mathfrak{m}_{\substack{r^{-1} \in R^{-1}_{mm_0}, mfm^{-1} \in MFM(transform(M,R'), R^{-1}_{mm_0})}}} mft(mfm^{-1}, r^{-1}) \\
\overset{\text{Def. sourceMatches...}}{=} \quad & \bigcup_{\mathfrak{m}_{\substack{r^{-1} \in R^{-1}_{mm_0}, mfm^{-1} \in srcMatches(transform(M,R'), R^{-1}_{mm_0})}}} mft(mfm^{-1}, r^{-1}) \\
\overset{\text{Def. 4.3.5 } (iii))}{=} \quad & \bigcup_{\mathfrak{m}_{\substack{r^{-1} \in R^{-1}_{mm_0}, mfm \in srcMatches(M,R')}}} mft(map(mfm), r^{-1}) \\
\overset{\text{Def. 4.3.5 (4.31)}}{=} \quad & \bigcup_{\mathfrak{m}_{\substack{mfm \in srcMatches(M,R')}}} mfm|_{mf} \\
\overset{\text{Def. 4.3.5 } (iv))}{=} \quad & m \in M : m_{mm} = mm_0
\end{aligned}
$$

❑

Beweis-Skizze zu Satz 4.3.2, Seite 170: Es wird im Folgenden gezeigt, dass Regelwerke, welche die in Satz 4.3.2 aufgestellten Forderungen erfüllen, Match-bijektiv sind.

Satz 4.3.2 (i) :
Entspricht Forderung (1)

Satz 4.3.2 (ii) :
Entspricht Forderung (1)

Satz 4.3.2 (iii) :
Gemäß Definition 4.3.5 muss hierzu eine bijektive Abbildung map mit den dort geforderten Eigenschaften existieren. Im Folgenden wird die Abbildung map für jede Regel $r \in \{r \in R : r|_{mv_0}|_{mm} = mm_0\}$ definiert als:

$$map(mfm) := mft(mfm, r)$$

Dann ist (4.30) trivialer Weise erfüllt:

$$mft(mfm, r) = map(mfm) = mft(mfm, r)(4.22gilt)$$
$$mft(mft(mfm, r), r^{-}1) = mfm$$

Für den Nachweis von (4.30) ist zu zeigen, dass gilt:

$$mft(mft(mfm, r), r^{-}1) = mfm|_{mf}$$

Gemäß Definition 3.5.5 für mft, Definition 3.5.2 von mfm und Definition 3.3.13 ist klar, dass gilt

$$mft(mft(mfm, r), r^{-1}) \cong mfm|_{mf}$$

D.h. die Modellfragmenttransformation durch die Umkehrregel erzeugt ein zum Ursprungsfragment kongruentes Modellfragment. Das so erzeugte Modellfragment ist identisch mit dem ursprünglichen Modellfragment, falls auch alle Werte von erzeugten Attributen und Identifikatoren mit dem ursprünglich gematchten Fragment übereinstimmen.

Dies ist der Fall, da sich diese Werte für beide Regeln aus demselben Gleichungssystem GL errechnen, wobei lediglich $match^1$ durch $match^0$ ersetzt wird und umgekehrt. Durch die Modellfragmenttransformation werden keine neuen Identifikatoren erzeugt, da der Wert $\diamondsuit$ gemäß (5a) nicht zugelassen ist. Wegen (1) sind die Gleichungssysteme jeweils in beide Richtung lösbar und haben gemäß (5b) immer genau eine Lösung, d.h. ihre Anwendung erzeugt niemals $(\emptyset, \emptyset)$ als Ergebnis. Demzufolge gilt auch Aussage (4.30) aus Definition 4.3.5.

Für den Nachweis, dass map tatsächlich bijektiv ist muss zudem gelten, dass die Modellfragmente des Urbildbereichs von map^{-1} genau dem Bildbereich von map entspricht. Dies entspricht der Aussage, dass durch Regeln des Regelwerks R, welche nicht mm_0 als Quellmetamodell haben, keine Elemente erzeugt werden können die durch das Umkehrregelwerk $R^{-1}_{mm_0}$ gematcht werden.

Besteht die Zielmodellvariable einer Regel r nur aus einer einzelnen Objektvariablen, so ist durch (3) sichergestellt, dass bei der Umkehrtransformation ausschliesslich die Regel r^{-1} die von ihr erzeugten Objekte matchen kann, da keine andere Regel eine Objektvariable dieses Typs enthalten darf.

Besteht die Zielmodellvariable einer Regel r aus mehreren Objektvariablen, so ist durch (4) sichergestellt, dass die Objektvariablen durch Objektvariablenassoziationen miteinander verbunden sind. Da jeder Typ von Objektvariablenassoziationen und Objektvariablen jeweils nur einmal vorkommen darf, kann jedem so erzeugten Fragment eindeutig die einzige Regel, die es erzeugt haben kann zugeordnet werden.

Satz 4.3.2 (iv) :
Die Regeln mit dem Quellmetamodell mm_0 erfassen jedes Quellmodell vollständig gemäß Forderung (6). Da die Regeln gemäß (5b) über keine Constraints verfügen dürfen, werden auch für alle Matches Modellfragmenttransformationen ausgeführt.

Folglich sind also jedes Regelwerk R, welches die in Satz gestellten Anforderungen erfüllt, streng Match-bijektiv.
$\overset{\text{Lemma 4.3.1}}{\Rightarrow}$ R ist streng bijektiv. ❑

A.2.4 Modell Rewriting-Transformationen

Beweis von Lemma 4.4.1, Seite 171: Zwei Objektvariablen ov_{10} und ov_{11} können lediglich dann nicht konfliktfrei sein, falls $ov_{10}|_{otv} = ov_{11}|_{otv}$ gilt. Dieser Fall ist jedoch in der Allgemeinheit nicht auszuschließen, die Konfliktfreiheit lässt sich jedoch ohne weiteres mit Hilfe von Lemma 4.1.6 (s. S. 122) nachweisen.
Annahme: $ov_{10}|_{otv} = ov_{11}|_{otv}$
1.Fall: $ov_{10}|_{otv}$ verfügt über Primärschlüssel:
Dann ergibt sich für $PCONF(ov_{10}, ov_{11}, k, l)$:

$$
\begin{aligned}
& & mfm_k^1|_{match_o}(ov_{10})|oi &= mfm_l^1|_{match_o}(ov_{11})|oi \wedge \\
GL'(ov_{10},k): & & mfm_k^1|_{match_o}(ov_{10})|oi &= pK(\{(n_i, mfm_k^0|_{match_o}(ov_{00}).n_i) : \\
& & & \qquad \exists (n_i,t_i) \in ov_{00}|_{otv}|_{Keys}\}\ ) \wedge \\
& \bigwedge_{n_i:(n_i,t_i)\in ov_{00}|_{otv}|_A} & mfm_k^1|_{match_o}(ov_{10}).n_i &= mfm_k^0|_{match_o}(ov_{00}).n_i \wedge \\
GL'(ov_{11},l): & & mfm_l^1|_{match_o}(ov_{11})|oi &= pK(\{(n_i, mfm_l^0|_{match_o}(ov_{01}).n_i) : \\
& & & \qquad \exists (n_i,t_i) \in ov_{01}|_{otv}|_{Keys}\}\ ) \wedge \\
& \bigwedge_{n_i:(n_i,t_i)\in ov_{01}|_{otv}|_A} & mfm_l^1|_{match_o}(ov_{11}).n_i &= mfm_l^0|_{match_o}(ov_{01}).n_i
\end{aligned}
$$

Das Gleichungssystem besitzt eine Lösung falls gilt:

$$
\begin{aligned}
& pK(\{(n_i, mfm_k^0|_{match_o}(ov_{00}).n_i) : \\
& \qquad \exists (n_i,t_i) \in ov_{00}|_{otv}|_{Keys}\}\ )
\end{aligned}
=
\begin{aligned}
& pK(\{(n_i, mfm_l^0|_{match_o}(ov_{01}).n_i) : \\
& \qquad \exists (n_i,t_i) \in ov_{01}|_{otv}|_{Keys}\}\ )
\end{aligned}
$$

Gemäß Definition 3.1.11 (3.11) (s. Seite 58) gilt diese Gleichung gdw.

$$
\begin{aligned}
& \forall (n_i,t_i) \in ov_{00}|_{otv}|_{Keys} = ov_{01}|_{otv}|_{Keys} : \\
& \quad ov_{00}|_V|(n,v) = ov_{01}|_V|(n,v)
\end{aligned}
$$

D.h. die beiden Objektvariablen können genau dann einen Konflikt verursachen, wenn sie auf Quellobjekte mit gleichen Primärschlüsselwerten matchen.
Für das Attribut-Gleichungssystem aus Definition 4.1.13 ergibt sich:

$$
\bigwedge_{att\in ov_{10}|_{otv}|_A|_n \setminus ov_{10}|_{otv}|_{Keys}|_n} mfm_k^1|_{match_o}(ov_{10}).att = mfm_l^1|_{match_o}(ov_{11}).att
$$

Dies entspricht der Forderung, dass auch sämtliche Nicht-Primärschlüsselattribute den gleichen Wert erhalten. Wegen Lemma 4.1.9 ist dies jedoch sichergestellt. Dieses besagt, dass Objekte des Quellmodells mit Identifikator auch generell identische Attributbutwerte enthalten. Da Objekte mit Identischen Primärschlüsselwerten immer identische Identifikatoren haben (sieh Def. 3.1.11 (3.11)) wird der Lösungsraum von *PCONF* also nicht weiter eingeschränkt.

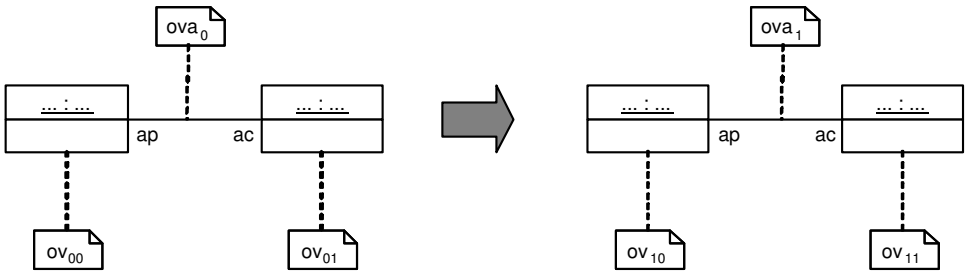


Abbildung A.1: Die Assoziationsregel $r_i a$ zur Klassenassoziation AE_i

2.Fall: $ov_{10}|_{otv}$ verfügt über keine Primärschlüssel:
Dann ergibt sich für $PCONF(ov_{10}, ov_{11}, k, l)$:

$$\begin{aligned}
& & mfm_k^1|_{match_o}(ov_{10})|oi &= mfm_l^1|_{match_o}(ov_{00})|oi \wedge \\
GL'(ov_{10},k): & & mfm_k^1|_{match_o}(ov_{10})|oi &= mfm_k^0|_{match_o}(ov_{00})|oi \wedge \\
& \bigwedge_{n_i:(n_i,t_i)\in ov_{00}|_{otv}|_A} & mfm_k^1|_{match_o}(ov_{10}).n_i &= mfm_k^0|_{match_o}(ov_{00}).n_i \ \wedge \\
GL'(ov_{11},l): & & mfm_l^1|_{match_o}(ov_{11})|oi &= mfm_l^0|_{match_o}(ov_{01})|oi \wedge \\
& \bigwedge_{n_i:(n_i,t_i)\in ov_{01}|_{otv}|_A} & mfm_l^1|_{match_o}(ov_{11}).n_i &= mfm_l^0|_{match_o}(ov_{01}).n_i
\end{aligned}$$

Das Gleichungssystem besitzt eine Lösung falls gilt:

$$mfm_k^0|_{match_o}(ov_{00})|oi = mfm_l^0|_{match_o}(ov_{01})|oi$$

D.h. die beiden Objektvariablen können genau dann einen Konflikt verursachen, wenn sie auf Quellobjekte mit gleichen Identifikatoren matchen.
Für das Attribut-Gleichungssystem aus Definition 4.1.13 ergibt sich:

$$\bigwedge_{att\in ov_{10}|_{otv}|_A|_n \setminus ov_{10}|_{otv}|_{Keys}|_n} mfm_k^1|_{match_o}(ov_{10}).att = mfm_l^1|_{match_o}(ov_{11}).att$$

Dies entspricht der Forderung, dass sämtliche Attribute (in diesem Fall existieren keine Primärschlüssel) den gleichen Wert erhalten. Wegen Lemma 4.1.9 ist dies jedoch sichergestellt. Dieses besagt, dass Objekte des Quellmodells mit gleichen Identifikator auch identische Attributbutwerte enthalten. ❑

Beweis von Lemma 4.4.2, Seite 171: Gemäß Definition 3.6.3 besteht das Regelwerk $R^{id}_{mm_0}$ aus je einer Regel für jede Klassenassoziation im Metamodell mm_0. O.B.d.A sei r_i die Regel zur Klassenassoziation AE_i in mm_0 wie in Abbildung A.1 dargestellt. Es wird nun im allgemeinen Fall die Anwendbarkeit einer Regel aus $R^{id}_{mm_0}$ gemäß Satz 4.1.6 (siehe S. 130) gezeigt. Hierzu sind drei Kriterien nachzuweisen:

Erzeugen gültiger Modellfragmente Die Lösungen des Gleichungssystems GL aus Definition 3.5.4 setzen sich aus Gleichungen der Form

$$\begin{aligned} & match^1_{\mathrm{o}}(ov_{10})|oi = match^0_{\mathrm{o}}(ov_{00})|oi \wedge \\ \bigwedge_{n_i:(n_i,t_i)\in ov_{00}|_{otv}|_A} & match^1_{\mathrm{o}}(ov_{10}).n_i = match^0_{\mathrm{o}}(ov_{00}).n_i \wedge \\ & match^1_{\mathrm{o}}(ov_{11})|oi = match^0_{\mathrm{o}}(ov_{01})|oi \wedge \\ \bigwedge_{n_i:(n_i,t_i)\in ov_{01}|_{otv}|_A} & match^1_{\mathrm{o}}(ov_{11}).n_i = match^0_{\mathrm{o}}(ov_{01}).n_i \end{aligned}$$

für Objektvariablen ohne Primärschlüssel und solchen der Form

$$\begin{aligned} & mfm^1_k|_{match_{\mathrm{o}}}(ov_{10})|oi = pK(\{(n_i, mfm^0_k|_{match_{\mathrm{o}}}(ov_{00}).n_i) : \\ & \qquad\qquad \exists(n_i,t_i) \in ov_{00}|_{otv}|_{Keys}\}) \wedge \\ \bigwedge_{n_i:(n_i,t_i)\in ov_{00}|_{otv}|_A} & mfm^1_k|_{match_{\mathrm{o}}}(ov_{10}).n_i = mfm^0_k|_{match_{\mathrm{o}}}(ov_{00}).n_i \wedge \\ & mfm^1_k|_{match_{\mathrm{o}}}(ov_{11})|oi = pK(\{(n_i, mfm^0_k|_{match_{\mathrm{o}}}(ov_{01}).n_i) : \\ & \qquad\qquad \exists(n_i,t_i) \in ov_{01}|_{otv}|_{Keys}\}) \wedge \\ \bigwedge_{n_i:(n_i,t_i)\in ov_{01}|_{otv}|_A} & mfm^1_k|_{match_{\mathrm{o}}}(ov_{11}).n_i = mfm^0_k|_{match_{\mathrm{o}}}(ov_{01}).n_i \end{aligned}$$

für Objektvariablen mit Primärschlüsseln zusammen. Diese bilden offensichtlich in jeder Kombination eine eindeutige Lösungen des Systems, d.h. Satz 4.1.1 (i) ist immer erfüllt. Sei nun r_j eine Assoziationsregel mit zwei Objektvariablen gleichen Typs ohne Primärschlüsselattribute auf der rechten Seite. Dann ergibt sich für $UGL_{validSrc}$ aus Definition 4.1.4:

$$\begin{aligned} & mfm^0_\mu|_{match_{\mathrm{o}}}(ov_{00})|_{oi} \neq mfm^0_\mu|_{match_{\mathrm{o}}}(ov_{01})|_{oi} \wedge \\ GL: \quad & \ldots \wedge \\ & match_{\mathrm{o}}{}^1_\mu(ov_{10})|oi = match_{\mathrm{o}}{}^0_\mu(ov_{00})|oi \\ & match_{\mathrm{o}}{}^1_\mu(ov_{11})|oi = match_{\mathrm{o}}{}^0_\mu(ov_{01})|oi \\ \Rightarrow \quad & match_{\mathrm{o}}{}^1_\mu(ov_{10})|oi \neq match_{\mathrm{o}}{}^1_\mu(ov_{11})|oi \end{aligned}$$

Offensichtlich gilt dann

$$\begin{aligned} & UGL_{validSrc} \wedge \\ match_{\mathrm{o}}{}^1_\mu(ov_{10})|oi \quad = \quad & match_{\mathrm{o}}{}^1_\mu(ov_{11})|oi \end{aligned}$$

hat keine Lösung, d.h. 4.1.1 (ii) ist für Objektvariablen ohne Primärschlüssel immer erfüllt. Die Überlegungen für Objektvariablen gleichen Typs mit Primärschlüssel verlaufen vollkommen analog und werden daher an dieser Stelle nicht weiter ausgeführt.

Deterministische Objektvariable Durch einsetzen lässt sich leicht zeigen:

$$\begin{aligned} dependsOn(ov_{10}, r_i) &= ov_{00} \\ \forall att \in ov_{10}|_{VV}|_a : attDependsOn(ov_{10}, a, r_i) &= ov_{00} \\ dependsOn(ov_{11}, r_i) &= ov_0 1 \\ \forall att \in ov_{11}|_{VV}|_a : attDependsOn(ov_{11}, a, r_i) &= ov_{01} \end{aligned}$$

$\Rightarrow$ Wegen $ov_{00} \overset{r_i|_{mv_0}}{\rightsquigarrow} ov_{00}$ und $ov_{01} \overset{r_i|_{mv_0}}{\rightsquigarrow} ov'_{01}$ gilt Satz 4.1.4 (4.15). D.h. sämtliche Objektvariable der rechten Regelseite von r_i sind deterministisch.

Konfliktfreie Objektvariable Sämtliche Objektvariablen eines Identitätsregelwerks sind gemäß Lemma 4.4.1 konfliktfrei.

Folglich sind sämtliche Regeln in beliebigen Identitäts-Regelwerken anwendbar. ❑

Beweis von Lemma 4.4.3, Seite 171:

Forderung 4.1.7 (i) Gilt gemäß Lemma 4.4.2.

Forderung 4.1.7 (ii) Forderung 4.1.7 (ii) besagt, dass sämtliche Objektvariablen gleichen Typs auf rechten Seiten von Regeln konfliktfrei sein müssen. Der Beweis dieser Eigenschaft verläuft aufgrund der gleichartigen Struktur aller Objektvariablen in $R^{id}_{mm_0}$ vollkommen analog zum Nachweis der gleichen Eigenschaft ((iii)) für die Anwendbarkeit einer einzelnen Regel und wird daher an dieser Stelle nicht weiter ausgeführt.

❑

Beweis von Lemma 4.4.4, Seite 171: Im Folgenden wird Satz 4.2.3 (s. S. 155) verwendet um die Obergrenzenkonformität von von Identitäts-Regelwerken zu beweisen. O.B.d.A sei $r_i \in R^{id}_{mm_0}$ eine Identitätsregel zur Klassenassoziation AE_i wie in Abbildung A.1 dargestellt. Zunächst wird untersucht ob die Objektvariablenassoziationen in R^A, deren Typ eine nicht-reflexive Klassenassoziation ist, redundant sein können. Gemäß Definition 3.6.3 existiert für jede Assoziation nur eine Objektvariablenassoziation auf der rechten Seite des Regelwerks. Definition 4.2.11 (ii) fordert aber für den Fall, dass $redundant(R^{id}_{mm_0}, r_a, ova_a)$ gilt unter anderem:

$$(i)\ \exists r_j \in R^{id}_{mm_0} : OVP(r_a, r_j) \neq \emptyset$$
$$(ii)\ \exists ova_j : ova_j = (ova_a|_{OVAT}, cardv_j, OVAE_j) \in r_j|_{mv_1}|_{OVA}$$

Aussage (ii) kann jedoch offensichtlich nicht für nicht-reflexive Assoziationen gelten. Daher kann mit Sicherheit ausgeschlossen werden, dass nicht-reflexive Assoziationen redundant sind. Also gilt:

$$\forall r_i \in R^{id}_{mm_0}, ova_j \in r_i|_{mv_1} : \neg redundant(R^{id}_{mm_0}, r_a, ova_a)$$

Weiterhin wissen wir:

$$depensOnId(ov_{10}) = ov_{00}$$
$$depensOnId(ov_{11}) = ov_{01}$$

$cannotCreateSame(ov_0, r_0), \ldots, (ov_n, r_n)$ liefert nur dann *true* falls ov_i, ov_j existieren mit $ov_i|otv \neq ov_j|otv$ und die beiden Objektvariablen ähnliche Identifikatoren haben. Dies ist für alle Objektvariablen der rechten Seite gleichen Typs wahr. Da jede Assoziation nur einmal mit einer Kardinalität von 1 auftritt gilt liefert $cannotCreateSame(ov_0, r_0), \ldots, (ov_n, r_n)$ den Wert true sobald sämtliche übergebenen Objektvariablen vom gleichen Typ sind.

Dementsprechend enthält $OV^{conf*}_{R^{id}_{mm_0}}$ alle Mengen von Objektvariablen der Rechten Seite die keine zwei Objektvariablen verschiedenen Typs enthalten. Hilfsaussage:
Generell gilt:

$$\max_{a \in A} a = \max_{A' \in \mathscr{P}(A)} \left(\max_{a \in A} a\right) \tag{A.11}$$

Für jede nicht-reflexive Assoziation ova zwischen den Objektvariablen ov_0 und ov_1 auf der rechten Seite ist dies: $\mathscr{P}(\{(r_c, r_c.ov_0), (r_a, r_a.ov_0)\})$ Für die Maximumsbildung gemäß Satz 4.2.3 reicht es wegen (A.11) die Menge $M_a = \{(r_c.ov_0), (r_a.ov_0)\}$ in Betracht zu ziehen.

Folglich ergibt sich für das im Zielmetamodell vorkommende Assoziationsende $ae_j \in AE_i$ an der Objektvariable ov_{11} mit Obergrenze *ub* durch einsetzen in Satz 4.2.3:

$$\begin{aligned}
&\max_{OV_k \in \{(r_a, ov_{10})\}} \{ \sum_{r_a, ova_a} ubVarCard(ova_a, ae_j, r_a) \\
&= \begin{cases} ova_1|_{cardv} & \text{falls } ov_{00} \overset{r_a|_{mv_0}}{\rightsquigarrow} ov_{01} \\ maxmatch(r_a|_{mv_0}, \{ov_{00}\}, \{ov_0 1\}) & \text{sonst} \end{cases} \\
&= \begin{cases} 1 & \text{falls } ov_{00} \overset{r_a|_{mv_0}}{\rightsquigarrow} ov_{01} \leq ub \\ \text{ub} \cdot ova_1|_{cardv} & \text{sonst} \end{cases}
\end{aligned}$$

Somit ist $R^{id}_{mm_0}$ obergrenzenkonform. ❑

Beweis von Lemma 4.4.5, Seite 172: O.B.d.A sei $r \in R^{id'}_{mm_0}$ eine C1-Identitätsregel wie in Abbildung A.1 dargestellt. Die jeweils einzige Objektvariablenassoziation auf beiden Seiten der Regel ist vom Typ $AE_i = \{ae_0, ae_1\}$ und ub_0, ub_1 sind die Multiplizitätsuntergrenzen von ae_0, bzw. ae_1.
Die Untergrenzenkonformität von *r* wird mit Hilfe von Satz 4.2.5 (s. S. 165) nachgewiesen. Es ergibt sich für

$$\begin{aligned}
dependsOn(ov_{10}) &= ov_{00} \\
dependsOn(ov_{11}) &= ov_{01}
\end{aligned}$$

Für *lbVarCard* ergibt sich dementsprechend:

$$\begin{aligned}
lbVarCard(ova, ae_1, r) &\overset{\text{Tab. 4.3}}{=} minmatch(mm_0, r|_{mv_0}, \{ov_0 0\}, \{ov_0 1\}) \\
&= ub_1 \\
lbVarCard(ova, ae_0, r) &\overset{\text{Tab. 4.3}}{=} minmatch(mm_0, r|_{mv_0}, \{ov_0 1\}, \{ov_0 0\}) \\
&= ub_0
\end{aligned}$$

Somit ist *r* gemäß Satz 4.2.5 untergrenzenkonform. Da dies im allgemeinen Fall gezeigt wurde gilt:

$$\forall r \in R^{id'}_{mm_0} : lbConform(r)$$

❑

Beweis von Lemma 4.4.6, Seite 172:

$$\begin{aligned}
&\forall r \in R^{id}_{mm_0} : lbConform(r) \quad \text{(Lemma 4.4.5)} \\
&\overset{\text{Satz 4.2.6}}{\Rightarrow} lbConform(R^{id}_{mm_0})
\end{aligned}$$

❑

Beweis von Satz 4.4.1, Seite 172: Der Beweis ergibt sich direkt aus Lemma 4.4.4, Lemma 4.4.6 und Satz 4.2.7 (siehe S. 166). ❑

Beweis von Satz 4.4.2, Seite 172: O.B.d.A sei $RW \in \mathbb{RRW}$ ein Rewrite-Regelwerk, M eine Menge von Quellmodellen, $mm_n \in \mathbb{MM}$ das Zielmetamodell und $(t_0, t_1, \ldots t_n)$ eine Ausführungsfolge. Weiterhin gelte die im Satz geforderte Eigenschaft. Für den Fall, dass die Ausführungsfolge die Länge 0 hat wird das unveränderte Zielmodell zurückgeliefert. Da dieses gemäß Voraussetzung metamodellkonform sein muss gilt die Aussage in diesem Fall trivialer Weise.

Es wird im Folgenden der Fall bewiesen, in dem die Ausführungsfolge eine Länge größer als 0 hat.

Beweis durch Induktion:

Induktionsanfang: Betrachtet wird das Modell m^0 nach Ausführung der Regeln mit $\{r \in RW : tx(RW,r) = t_0\}$: Gemäß Definition 3.6.5 gilt:

$$m^0 = transform(M, R^{id}_{mm} \cup \{r \in RW : tx(RW,r) = t_0\})$$

und

$$rwTransform(M, RW, (t_0, t1, \ldots t_n)) = rwTransform\Big(m^0 \cup \{m \in M : m_{mm} \neq mm_n\}, RW, (t_1, \ldots, t_n)\Big)$$

Da gemäß der Voraussetzung das Regelwerk $R^{id}_{mm} \cup \{r \in RW : tx(RW,r) = t_0\}$ metamodellkonform ist ist auch das Modell m^0 konform zu seinem Metamodell (siehe Def. 4.4.1). Da das Zielmodell ein C1-Modell ist und sämtliche Regeln gemäß Definition 3.6.4 (3.26) C1-Regeln sind ist das Modell m^0 gemäß Lemma 3.6.1 ebenfalls wieder ein C1-Modellfragment.

Induktionsannahme: Das Modell m^{i-1} nach Ausführung der Folge $(t_0, \ldots, t_{i-1})$ ist metamodellkonform.

Induktionsschritt: Betrachtet wird das Modell m^i nach Ausführung der Folge $(t_0, \ldots, t_i)$. Gemäß Definition 3.6.5 gilt:

$$m^i = transform(m^{i-1} \cup \{m \in M : m_{mm} \neq mm_n\}, R^{id}_{mm} \cup \{r \in RW : tx(RW,r) = t_i\})$$

Analog zum Induktionsanfang muss m^i metamodellkonform sein und $C1(mm^i)$ gelten, da auch das Regelwerk $R^{id}_{mm} \cup \{r \in RW : tx(RW,r) = t_i\}$ gemäß Annahme metamodellkonform ist.

Folglich werden durch beliebige Ausführungsfolgen von RW lediglich metamodellkonforme Modelle erzeugt. ❑

Beweis von Lemma 4.4.7, Seite 172: Gemäß Satz 4.1.7 ist ein Regelwerk anwendbar, falls jede Regel anwendbar ist und für jedes Paar ov_i, ov_j aus Objektvariablen der rechten Regelseiten gilt $conflictFree(ov_i, ov_j)$.

Die Anwendbarkeit der Regeln aus dem Regelwerk $R^{id}_{mm_n}$ wurde bereits durch Lemma 4.4.3 nachgewiesen. Die Anwendbarkeit der Regeln aus RW^i ist Teil der Forderung von Lemma 4.4.7.

Die Konfliktfreiheit aller Zielobjektvariablen des Identitätsregelwerks wurde bereits durch Lemma 4.4.1 nachgewiesen. Dementsprechend müssen für den Nachweise der Anwendbarkeit des Regelwerks $R^{id}_{mm_n} \cup RW^i$ lediglich die in Forderung (ii) definierten Paare von Objektvariablen überprüft werden. ❑

B Metamodellkonformität des Regelwerks zur Integration von E/R-Diagrammen

B.1 Anwendbarkeit des Regelwerks

Im Rahmen dieses Abschnitts wird die Anwendbarkeit des Regelwerks R_{ER} verifiziert. Hierzu wird zunächst die Anwendbarkeit jeder einzelnen Regel nachgewiesen.

B.1.1 Anwendbarkeit von r_1

Erzeugen gültiger Modellfragmente

Für das Gleichungssystem *GL* ergibt sich:

$$\begin{aligned} mfm^0_\mu|_{match_o}(ev).name &= n \\ mfm^1_\mu|_{match_o}(cv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(cv).name)\}) \\ mfm^1_\mu|_{match_o}(cv).name &= n \end{aligned}$$

Demnach ergibt sich als einzige mögliche Lösung für *GL*:

$$\begin{aligned} mfm^1_\mu|_{match_o}(cv)|_{oi} &= pk(\{(name, mfm^0_\mu|_{match_o}(ev).name)\}) \\ mfm^1_\mu|_{match_o}(cv).name &= mfm^0_\mu|_{match_o}(ev).name \end{aligned}$$

Auf der rechten Regelseite existieren keine Objektvariablen gleichen Typs. Demnach gilt gemäß Satz 4.1.2 (S. 115):

$$createsValidFragments(r_1)$$

Deterministische Objektvariable

Zu überprüfen ist lediglich die einzige Objektvariable *cv*. Diese verfügt lediglich über ein Attribut, welches ein Primärschlüsselattribut ist. Gemäß Satz 4.1.4 (4.14) (S. 119) gilt:

$$deterministic(r_1, cv)$$

Konfliktfreie Objektvariable

Da auf der rechten Regelseite lediglich eine Objektvariable existiert können keine Konflikte zwischen verschiedenen Objektvariablen der rechten Seite auftreten.

$\Rightarrow r_1$ ist anwendbar.

B.1.2 Anwendbarkeit von r_2

Erzeugen gültiger Modellfragmente

Für das Gleichungssystem *GL* ergibt sich:

$$\begin{aligned}
mfm^0_\mu|_{match_o}(ev).name &= cName \\
mfm^0_\mu|_{match_o}(aev)|_{oi} &= attId \\
mfm^0_\mu|_{match_o}(aev).name &= attName \\
mfm^0_\mu|_{match_o}(aev).mehrfach &= \text{„}false\text{“} \\
mfm^0_\mu|_{match_o}(wv).name &= typeName \\
mfm^1_\mu|_{match_o}(cv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(cv).name)\}) \\
mfm^1_\mu|_{match_o}(cv).name &= cName \\
mfm^1_\mu|_{match_o}(acv)|_{oi} &= attId \\
mfm^1_\mu|_{match_o}(acv).name &= attName \\
mfm^1_\mu|_{match_o}(tv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(tv).name)\}) \\
mfm^1_\mu|_{match_o}(tv).name &= typeName
\end{aligned}$$

Demnach ergibt sich als einzige mögliche Lösung für *GL*:

$$\begin{aligned}
mfm^1_\mu|_{match_o}(cv)|_{oi} &= pk(\{(name, mfm^0_\mu|_{match_o}(ev).name)\}) \\
mfm^1_\mu|_{match_o}(cv).name &= mfm^0_\mu|_{match_o}(ev).name \\
mfm^1_\mu|_{match_o}(acv)|_{oi} &= mfm^0_\mu|_{match_o}(aev)|_{oi} \\
mfm^1_\mu|_{match_o}(acv).name &= mfm^0_\mu|_{match_o}(aev).name \\
mfm^1_\mu|_{match_o}(tv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(tv).name)\}) \\
mfm^1_\mu|_{match_o}(tv).name &= mfm^1_\mu|_{match_o}(tv).name
\end{aligned}$$

Auf der rechten Regelseite existieren keine Objektvariablen gleichen Typs. Demnach gilt gemäß Satz 4.1.2 (S. 115):

$$createsValidFragments(r_2)$$

Deterministische Objektvariable

Zu überprüfen sind Objektvariablen *cv*, *acv* und *tv*:

cv verfügt lediglich über ein Attribut, welches ein Primärschlüsselattribut ist. Gemäß Satz 4.1.4 (4.14) (S. 119) gilt:

$$deterministic(r_2, cv)$$

acv verfügt über eine Attributvariable *name*. Es gilt:

$$
\begin{array}{ll}
 & dependsOn(acv, r_2) = aev \\
 & attDependsOn(acv, name, r_2) = aev \\
aev \overset{r_2|_{mv_0}}{\rightsquigarrow} aev & \\
\Rightarrow & dependsOn(acv, r_2) \\
 & \overset{r_2|_{mv_0}}{\rightsquigarrow} attDependsOn(acv, name, r_2) \\
\overset{\text{Satz 4.1.4 (4.15)}}{\Rightarrow} & deterministic(r_2, acv)
\end{array}
$$

tv verfügt lediglich über ein Attribut, welches ein Primärschlüsselattribut ist. Gemäß Satz 4.1.4 (4.14) (S. 119) gilt:

$$deterministic(r_2, tv)$$

Konfliktfreie Objektvariable

Für jedes Paar aus Objektvariablen ov_i, ov_j der rechten Regelseite gilt:

$$
\begin{array}{ll}
 & ov_i|_{otv} \neq ov_j|_{otv} \\
\overset{\text{Lemma 4.1.5}}{\Rightarrow} & \neg potConflicting(r_2, ov_i, ov_j) \\
\overset{\text{Lemma 4.1.6}}{\Rightarrow} & conflictFree(r_2, ov_i, ov_j)
\end{array}
$$

$\Rightarrow r_2$ ist anwendbar.

B.1.3 Anwendbarkeit von r_3

Erzeugen gültiger Modellfragmente

Für das Gleichungssystem *GL* ergibt sich:

$$
\begin{aligned}
mfm^0_\mu|_{match_o}(ev).name &= cName \\
mfm^0_\mu|_{match_o}(aev)|_{oi} &= attId \\
mfm^0_\mu|_{match_o}(aev).name &= attName \\
mfm^0_\mu|_{match_o}(aev).optional &= \text{„}true\text{“} \\
mfm^0_\mu|_{match_o}(aev).mehrfach &= \text{„}true\text{“} \\
mfm^0_\mu|_{match_o}(wv).name &= typeName \\
mfm^1_\mu|_{match_o}(c0v)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(c0v).name)\}) \\
mfm^1_\mu|_{match_o}(c0v).name &= cName \\
mfm^1_\mu|_{match_o}(cae0v)|_{oi} &= genId(\ldots) \\
mfm^1_\mu|_{match_o}(cae0v).roleName &= cName \\
mfm^1_\mu|_{match_o}(cae0v).aggregationType &= \text{„}composition\text{“} \\
mfm^1_\mu|_{match_o}(cae0v).multiplicity &= \text{„}(1,1)\text{“} \\
mfm^1_\mu|_{match_o}(cae0v).isNavigable &= \text{„}true\text{“} \\
mfm^1_\mu|_{match_o}(cav)|_{oi} &= genId(\ldots) \\
mfm^1_\mu|_{match_o}(cae1v)|_{oi} &= genId(\ldots) \\
mfm^1_\mu|_{match_o}(cae1v).roleName &= attName \\
mfm^1_\mu|_{match_o}(cae1v).aggregationType &= \text{„}none\text{“} \\
mfm^1_\mu|_{match_o}(cae1v).multiplicity &= \text{„}(0,*)\text{“} \\
mfm^1_\mu|_{match_o}(cae1v).isNavigable &= \text{„}true\text{“} \\
mfm^1_\mu|_{match_o}(c1v)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(c1v).name)\}) \\
mfm^1_\mu|_{match_o}(c1v).name &= cName + attName \\
mfm^1_\mu|_{match_o}(av)|_{oi} &= attId \\
mfm^1_\mu|_{match_o}(av).name &= \text{„}value\text{“} \\
mfm^1_\mu|_{match_o}(tv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(tv).name)\}) \\
mfm^1_\mu|_{match_o}(tv).name &= typeName
\end{aligned}
$$

Demnach ergibt sich als einzige mögliche Lösung für GL:

$$
\begin{aligned}
mfm^1_\mu|_{match_o}(c0v)|_{oi} &= pk(\{(name, mfm^0_\mu|_{match_o}(ev).name)\}) \\
mfm^1_\mu|_{match_o}(c0v).name &= mfm^0_\mu|_{match_o}(ev).name \\
mfm^1_\mu|_{match_o}(cae0v)|_{oi} &= genId(...) \\
mfm^1_\mu|_{match_o}(cae0v).roleName &= cName \\
mfm^1_\mu|_{match_o}(cae0v).aggregationType &= \text{„}composition\text{“} \\
mfm^1_\mu|_{match_o}(cae0v).multiplicity &= \text{„}(1,1)\text{“} \\
mfm^1_\mu|_{match_o}(cae0v).isNavigable &= \text{„}true\text{“} \\
mfm^1_\mu|_{match_o}(cav)|_{oi} &= genId(...) \\
mfm^1_\mu|_{match_o}(cae1v)|_{oi} &= genId(...) \\
mfm^1_\mu|_{match_o}(cae1v).roleName &= mfm^0_\mu|_{match_o}(aev).name \\
mfm^1_\mu|_{match_o}(cae1v).aggregationType &= \text{„}none\text{“} \\
mfm^1_\mu|_{match_o}(cae1v).multiplicity &= \text{„}(0,*)\text{“} \\
mfm^1_\mu|_{match_o}(cae1v).isNavigable &= \text{„}true\text{“} \\
mfm^1_\mu|_{match_o}(c1v)|_{oi} &= pk(\{(name, mfm^0_\mu|_{match_o}(ev).name \\
&\quad + mfm^0_\mu|_{match_o}(aev).name)\}) \\
mfm^1_\mu|_{match_o}(c1v).name &= mfm^0_\mu|_{match_o}(ev).name \\
&\quad + mfm^0_\mu|_{match_o}(aev).name \\
mfm^1_\mu|_{match_o}(av)|_{oi} &= mfm^1_\mu|_{match_o}(av)|_{oi} \\
mfm^1_\mu|_{match_o}(av).name &= \text{„}value\text{“} \\
mfm^1_\mu|_{match_o}(tv)|_{oi} &= pk(\{(name, mfm^0_\mu|_{match_o}(wv).name)\}) \\
mfm^1_\mu|_{match_o}(tv).name &= mfm^0_\mu|_{match_o}(wv).name
\end{aligned}
$$

$\Rightarrow$ Satz 4.1.1 (i) ist erfüllt.

Untersucht werden nun alle Objektvariablen der rechten Regelseite mit gleichem Typ. Das Ungleichungsystem $UGL_{validSrc}$ ist hierbei leer, da die linke Regelseite keine Objektvariablen gleichen Typs enthält (siehe Def. 4.1.4, S. 112). dem leeren

Für $c0v$ und $cv1$ ergibt sich:

$$
\begin{aligned}
& GL && \wedge \\
& mfm^1_\mu|_{match_o}(c0v)|_{oi} && = mfm^1_\mu|_{match_o}(c1v)|_{oi} \\
\Rightarrow\ & pk(\{(name, mfm^0_\mu|_{match_o}(ev).name)\}) && = pk(\{(name, mfm^0_\mu|_{match_o}(ev).name \\
& && \quad + mfm^0_\mu|_{match_o}(aev).name)\})
\end{aligned}
$$

$\Rightarrow \lightning$ wegen Def. pK, siehe Def. 3.1.11, S. 58

Für *cae*0*v* und *cae*1*v* ergibt sich:

$$\begin{array}{rll} & GL & \wedge \\ & mfm^1_\mu|_{match_o}(cae0v)|_{oi} & = mfm^1_\mu|_{match_o}(cae1v)|_{oi} \\ \Rightarrow & genId(\ldots) & = genId(\ldots) \end{array}$$

$\Rightarrow \lightning$ wegen Def. *genId*, siehe Def. 3.5.4, S. 86

$$\overset{\text{Satz 4.1.1 } (ii)}{\Rightarrow} createsValidFragments(r_3)$$

Deterministische Objektvariable

Zu überprüfen sind Objektvariablen *c*0*v*, *cae*0*v*, *cav*, *cae*1*v*, *c*1*v*, *av* und *tv*:

*c*0*v* verfügt lediglich über ein Attribut, welches ein Primärschlüsselattribut ist. Gemäß Satz 4.1.4 (4.14) (S. 119) gilt:

$$deterministic(r_3, c0v)$$

*cae*0*v* hat den Wert $\diamondsuit$ als Identifikatorterm. Gemäß Satz 4.1.4 (4.13) gilt:

$$deterministic(r_3, cae0v)$$

cav hat den Wert $\diamondsuit$ als Identifikatorterm. Gemäß Satz 4.1.4 (4.13) gilt:

$$deterministic(r_3, cav)$$

*cae*1*v* hat den Wert $\diamondsuit$ als Identifikatorterm. Gemäß Satz 4.1.4 (4.13) gilt:

$$deterministic(r_3, cae1v)$$

*c*1*v* verfügt lediglich über ein Attribut, welches ein Primärschlüsselattribut ist. Gemäß Satz 4.1.4 (4.14) (S. 119) gilt:

$$deterministic(r_3, c1v)$$

av hat lediglich ein Attribut *name*. Da dieses den konstanten Wert „value“ hat gilt gemäß 4.1.4 (4.14)

$$deterministic(r_3, av)$$

tv verfügt lediglich über ein Attribut, welches ein Primärschlüsselattribut ist. Gemäß Satz 4.1.4 (4.14) (S. 119) gilt:

$$deterministic(r_3, tv)$$

Konfliktfreie Objektvariable

Untersucht werden alle Paare von Objektvariablen der rechten Regelseite mit gleichem Typ.

Für das Paar aus deb Objektvariablen $cae0v$ und $cae1v$ der rechten Regelseite gilt:

$$\begin{aligned} & \diamondsuit = cae0v|_{oiv} \not\sim cae1v|_{oiv} = \diamondsuit \\ \overset{\text{Lemma 4.1.5}}{\Rightarrow} & \neg potConflicting(r_3, cae0v, cae1v) \\ \overset{\text{Lemma 4.1.6}}{\Rightarrow} & conflictFree(r_3, cae0v, cae1v) \end{aligned}$$

Für das Paar aus deb Objektvariablen $c0v$ und $c1v$ der rechten Regelseite gilt:

$$\begin{aligned} & \{n : \exists c0v|_{otv}|_A \ni (n,t) \notin c0v|_{otv}|_{Keys}\} = \emptyset \\ \overset{\text{Lemma 4.1.5}}{\Rightarrow} & \neg potConflicting(r_3, cae0v, cae1v) \\ \overset{\text{Lemma 4.1.6}}{\Rightarrow} & conflictFree(r_3, cae0v, cae1v) \end{aligned}$$

$\Rightarrow r_3$ ist anwendbar.

B.1.4 Anwendbarkeit von r_4

Der Nachweis der Anwendbarkeit von Regel „ER-RAM Rule 4“ verläuft vollkommen analog zu dem für Regel „ER-RAM Rule 3“ und wird daher an dieser Stelle nicht weiter ausgeführt.

B.1.5 Anwendbarkeit von r_5

Erzeugen gültiger Modellfragmente

Für das Gleichungssystem GL ergibt sich:

$$\begin{aligned} mfm^0_\mu|_{match_o}(ev).name &= cName \\ mfm^0_\mu|_{match_o}(aev)|_{oi} &= attId \\ mfm^0_\mu|_{match_o}(aev).optional &= \text{„}false\text{“} \\ mfm^0_\mu|_{match_o}(aev).mehrfach &= \text{„}false\text{“} \\ mfm^0_\mu|_{match_o}(wv).name &= typeName \\ mfm^1_\mu|_{match_o}(cv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(cv).name)\}) \\ mfm^1_\mu|_{match_o}(cv).name &= cName \\ mfm^1_\mu|_{match_o}(acv)|_{oi} &= attId \\ mfm^1_\mu|_{match_o}(acv).name &= \diamond \\ mfm^1_\mu|_{match_o}(tv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(tv).name)\}) \\ mfm^1_\mu|_{match_o}(tv).name &= typeName \end{aligned}$$

Demnach ergibt sich als einzige mögliche Lösung für *GL*:

$$\begin{aligned}
mfm^1_\mu|_{match_o}(cv)|_{oi} &= pk(\{(name, mfm^0_\mu|_{match_o}(ev).name)\}) \\
mfm^1_\mu|_{match_o}(cv).name &= mfm^0_\mu|_{match_o}(ev).name \\
mfm^1_\mu|_{match_o}(acv)|_{oi} &= mfm^0_\mu|_{match_o}(aev)|_{oi} \\
mfm^1_\mu|_{match_o}(acv).name &= \diamond \\
mfm^1_\mu|_{match_o}(tv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(tv).name)\}) \\
mfm^1_\mu|_{match_o}(tv).name &= mfm^1_\mu|_{match_o}(tv).name
\end{aligned}$$

Auf der rechten Regelseite existieren keine Objektvariablen gleichen Typs. Demnach gilt gemäß Satz 4.1.2 (S. 115):

$$createsValidFragments(r_5)$$

Deterministische Objektvariable

Zu überprüfen sind Objektvariablen *cv*, *acv* und *tv*. Der Nachweis, dass alle diese Objektvariablen deterministisch sind verläuft vollkommen analog zu dem entsprechenden Nachweis für Regel r_2.

Konfliktfreie Objektvariable

Für jedes Paar aus Objektvariablen ov_i, ov_j der rechten Regelseite gilt:

$$\begin{aligned}
& ov_i|_{otv} \neq ov_j|_{otv} \\
\overset{\text{Lemma 4.1.5}}{\Rightarrow} \; & \neg potConflicting(r_2, ov_i, ov_j) \\
\overset{\text{Lemma 4.1.6}}{\Rightarrow} \; & conflictFree(r_5, ov_i, ov_j)
\end{aligned}$$

$\Rightarrow r_5$ ist anwendbar.

B.1.6 Anwendbarkeit von r_6

Erzeugen gültiger Modellfragmente

Für das Gleichungssystem *GL* ergibt sich:

$$
\begin{aligned}
mfm^0_\mu|_{match_o}(ev).name &= cName \\
mfm^0_\mu|_{match_o}(bev)|_{oi} &= endId \\
mfm^0_\mu|_{match_o}(bev).name &= endName \\
mfm^0_\mu|_{match_o}(aev).optional &= endOpt \\
mfm^0_\mu|_{match_o}(aev).mehrfach &= endCard \\
mfm^0_\mu|_{match_o}(bv)|_{oi} &= assoId \\
mfm^1_\mu|_{match_o}(cv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(cv).name)\}) \\
mfm^1_\mu|_{match_o}(cv).name &= cName \\
mfm^1_\mu|_{match_o}(caev)|_{oi} &= endId \\
mfm^1_\mu|_{match_o}(caev).roleName &= endName \\
mfm^1_\mu|_{match_o}(caev).aggregationType &= \text{„}none\text{“} \\
mfm^1_\mu|_{match_o}(caev).multiplicity &= (endOpt?0:1, endCard?*:1) \\
mfm^1_\mu|_{match_o}(caev).isNavigable &= \text{„}true\text{“} \\
mfm^1_\mu|_{match_o}(cav)|_{oi} &= assoId
\end{aligned}
$$

Demnach ergibt sich als einzige mögliche Lösung für *GL*:

$$
\begin{aligned}
mfm^1_\mu|_{match_o}(cv).name &= mfm^0_\mu|_{match_o}(ev).name \\
mfm^1_\mu|_{match_o}(caev)|_{oi} &= mfm^0_\mu|_{match_o}(bev)|_{oi} \\
mfm^1_\mu|_{match_o}(caev).roleName &= mfm^0_\mu|_{match_o}(bev).name \\
mfm^1_\mu|_{match_o}(caev).aggregationType &= \text{„}none\text{“} \\
mfm^1_\mu|_{match_o}(caev).multiplicity &= (mfm^0_\mu|_{match_o}(aev).optional?0:1, \\
&\qquad mfm^0_\mu|_{match_o}(aev).mehrfach?*:1) \\
mfm^1_\mu|_{match_o}(caev).isNavigable &= \text{„}true\text{“} \\
mfm^1_\mu|_{match_o}(cav)|_{oi} &= mfm^0_\mu|_{match_o}(bv)|_{oi}
\end{aligned}
$$

Auf der rechten Regelseite existieren keine Objektvariablen gleichen Typs. Demnach gilt gemäß Satz 4.1.2 (S. 115):

$$createsValidFragments(r_6)$$

Deterministische Objektvariable

Zu überprüfen sind Objektvariablen *cv*, *caev* und *cav*:

cv verfügt lediglich über ein Attribut, welches ein Primärschlüsselattribut ist. Gemäß Satz 4.1.4 (4.14) (S. 119) gilt:

$$deterministic(r_6, c0v)$$

caev verfügt über eine Attributvariablen *roleName*, *aggregationType*, *multiplicity* und *isNavigable*. Es gilt:

$$
\begin{aligned}
& dependsOn(caev, r_6) = \{bev\} \\
& attDependsOn(caev, roleName, r_6) = \{bev\} \\
\{bev\} \overset{r_6|mv_0}{\rightsquigarrow} \{bev\} \Rightarrow\ & dependsOn(caev, r_6) \\
& \overset{r_6|mv_0}{\rightsquigarrow} attDependsOn(caev, roleName, r_6) \\
\wedge & \\
& attDependsOn(caev, aggregationType, r_6) = \emptyset \\
\{bev\} \overset{r_6|mv_0}{\rightsquigarrow} \emptyset \Rightarrow\ & dependsOn(caev, r_6) \\
& \overset{r_6|mv_0}{\rightsquigarrow} attDependsOn(caev, roleName, r_6) \\
\wedge & \\
& attDependsOn(caev, multiplicity, r_6) = \{bev\} \\
\{bev\} \overset{r_6|mv_0}{\rightsquigarrow} \{bev\} \Rightarrow\ & dependsOn(caev, r_6) \\
& \overset{r_6|mv_0}{\rightsquigarrow} attDependsOn(caev, multiplicity, r_6) \\
\wedge & \\
& attDependsOn(caev, isNavigable, r_6) = \emptyset \\
\{bev\} \overset{r_6|mv_0}{\rightsquigarrow} \emptyset \Rightarrow\ & dependsOn(caev, r_6) \\
& \overset{r_6|mv_0}{\rightsquigarrow} attDependsOn(caev, isNavigable, r_6) \\
\overset{\text{Satz 4.1.4 (4.15)}}{\Rightarrow}\ & deterministic(r_6, caev)
\end{aligned}
$$

cav verfügt über keinerlei Attribute

$$
\overset{\text{Aussagen über } \emptyset\text{, Satz 4.1.4 (4.14)}}{\Rightarrow} deterministic(r_6, cav)
$$

Konfliktfreie Objektvariable

Für jedes Paar aus Objektvariablen ov_i, ov_j der rechten Regelseite gilt:

$$
\begin{aligned}
& ov_i|_{otv} \neq ov_j|_{otv} \\
\overset{\text{Lemma 4.1.5}}{\Rightarrow}\ & \neg potConflicting(r_2, ov_i, ov_j) \\
\overset{\text{Lemma 4.1.6}}{\Rightarrow}\ & conflictFree(r_5, ov_i, ov_j)
\end{aligned}
$$

$\Rightarrow r_5$ ist anwendbar.

B.1.7 Anwendbarkeit von R_{ER}

Der Nachweis der Anwendbarkeit des Regelwerks R_{ER} erfolgt anhand von Satz 4.1.7. Aussage (i) des Satzes wurde bereits in den vorangegangenen Abschnitten bewiesen. Es bleibt nun zu zeigen, dass

sämtliche Paare aus Objektvariablen gleichen Typs aus Zielmodellvariablen des Regelwerks konfliktfrei sind. Da dies für Objektvariablen aus derselben Regel bereits im Rahmen des Nachweises der Anwendbarkeit der einzelnen Regeln geschehen ist muss nunmehr nur noch die Konfliktfreiheit von Objektvariablen aus verschiedenen Regeln bewiesen werden.

Nachweis für Objektvariable vom Typ Class oder Type:

Es ex. nur ein Primärschlüsselattribut. D.h. Lemma 4.1.6 macht eine Aussage über $\emptyset$
$\Rightarrow$ OK.

Nachweis für die übrigen Objektvariablen: Tabelle B.1 fasst den Nachweis für die übrigen Paare von Objektvariablen aus unterschiedlichen Regeln zusammen.

$$\Longrightarrow R_{ER} \text{ ist anwendbar.} \tag{B.1}$$

B.2 Obergrenzenkonformität

Um Assoziationen, Objektvariablenassoziationen und deren Enden für den Nachweis der Metamodellkonformität referenzieren zu können, wird zunächst eine Reihe von Kurzschreibweisen eingeführt. Die Tabellen B.2 und B.3 fassen diese zusammen.

Redundante Objektvariablenassoziationen

Innerhalb der Regelwerks existiert lediglich

$$r_2|_{mv_0} \sqsubseteq r_5|_{mv_0}$$

$$\begin{aligned} corresponds_0(r_2.ev) &= r_5.ev \\ corresponds_0(r_2.aev) &= r_5.aev \\ corresponds_0(r_2.wv) &= r_5.wv \end{aligned}$$

Eine mögliche Lösung für die Menge OVP lautet:

$$OVP(r_5, r_2) = \{\{(r_5.cv, r_2.cv), (r_5.acv, r_2.acv), (r_5.tv, r_2.tv)\}\}$$

Für die Lösungen der Objektidentifikatoren der Rechten Regelseite gilt:
Lösungen für die Identifikatoren der rechten Regelseite in r_2:

$$\begin{aligned} mfm^1_\mu|_{match_o}(cv)|_{oi} &= pk(\{(name, mfm^0_\mu|_{match_o}(ev).name)\}) \\ mfm^1_\mu|_{match_o}(acv)|_{oi} &= mfm^0_\mu|_{match_o}(aev)|_{oi} \\ mfm^1_\mu|_{match_o}(tv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(tv).name)\}) \end{aligned}$$

Lösungen für die Identifikatoren der rechten Regelseite in r_5:

$$\begin{aligned} mfm^1_\mu|_{match_o}(cv)|_{oi} &= pk(\{(name, mfm^0_\mu|_{match_o}(ev).name)\}) \\ mfm^1_\mu|_{match_o}(acv)|_{oi} &= mfm^0_\mu|_{match_o}(aev)|_{oi} \\ mfm^1_\mu|_{match_o}(tv)|_{oi} &= pk(\{(name, mfm^1_\mu|_{match_o}(tv).name)\}) \end{aligned}$$

ov_i	ov_j	$conflictFree(R_{ER}, ov_i, ov_j)$?
$r_2.acv$	$r_3.acv$	$r_2.acv\|_{oiv} \not\sim r_3.acv\|_{oiv}$ $\overset{\text{Lemma 4.1.5}}{\Rightarrow} \neg potConflicting(R_{ER}, r_2.acv, r_3.acv)$ $\overset{\text{Lemma 4.1.6}}{\Rightarrow} conflictFree(R_{ER}, r_2.acv, r_3.acv)$
$r_2.acv$	$r_4.acv$	– " –
$r_3.acv$	$r_4.acv$	– " –
$r_2.acv$	$r_5.acv$	$r_5.acv.name = \diamondsuit$ $\overset{\text{Lemma 4.1.6}}{\Rightarrow} conflictFree(R_{ER}, r_2.acv, r_5.acv)$
$r_3.acv$	$r_5.acv$	– " –
$r_4.acv$	$r_5.acv$	– " –
$r_3.cae0v$	$r_4.cae0v$	$r_3.cae0v\|_{oiv} \not\sim r_4.cae0v\|_{oiv}$ $\overset{\text{Lemma 4.1.5}}{\Rightarrow} \neg potConflicting(R_{ER}, r_3.cae0v, r_4.cae0v)$ $\overset{\text{Lemma 4.1.6}}{\Rightarrow} conflictFree(R_{ER}, r_3.cae0v, r_.cae0v)$
$r_3.cae0v$	$r_4.cae1v$	– " –
$r_3.cae1v$	$r_4.cae0v$	– " –
$r_3.cae1v$	$r_4.cae1v$	– " –
$r_3.cae0v$	$r_6.caev$	– " –
$r_3.cae1v$	$r_6.caev$	– " –
$r_4.cae0v$	$r_6.caev$	– " –
$r_4.cae1v$	$r_6.caev$	– " –
$r_3.cav$	$r_4.cav$	$r_3.cav\|_{oiv} \not\sim r_4.cav\|_{oiv}$ $\overset{\text{Lemma 4.1.5}}{\Rightarrow} \neg potConflicting(R_{ER}, r_3.cav, r_4.cav)$ $\overset{\text{Lemma 4.1.6}}{\Rightarrow} conflictFree(R_{ER}, r_3.cav, r_4.cav)$
$r_3.cav$	$r_6.cav$	– " –
$r_4.cav$	$r_6.cav$	– " –

Tabelle B.1: Nachweis der Konfliktfreiheit von Objektvariablen aus unterschiedlichen Regeln von R_{ER}

Name	Bezeichnet
CA	Klassenassoziation „class-att“ zwischen *Class* und *Attribute* im RAM-Metamodell
CA_c	Klassenassoziationsende bei *Class* von CA
CK	Klassenassoziation „class-key“ zwischen *Class* und *Attribute* im RAM-Metamodell
CK_c	Klassenassoziationsende bei *Class* von CA
AT	Klassenassoziation „att-isOfType“ zwischen *Attribute* und *Type* im RAM-Metamodell
AT_t	Klassenassoziationsende bei *Type* von AT
CE	Klassenassoziation „class-end“ zwischen *Class* und *ClassAssociationEnd* im RAM-Metamodell
CE_c	Klassenassoziationsende bei *Class* von CE
EA	Klassenassoziation „end-association“ zwischen *ClassAssociationEnd* und *ClassAssociation* im RAM-Metamodell
EA_e	Klassenassoziationsende bei *ClassAssociationEnd* von EA
EA_a	Klassenassoziationsende bei *ClassAssociation* von EA
$r_2.ova_{ca}$	Objektvariablenassoziation zwischen $r_2.cv$ und $r_2.acv$
$r_2.ae_{cca}$	Objektvariablenassoziationsende an $r_2.cv$ von $r_2.ova_{ca}$
$r_2.ova_{at}$	Objektvariablenassoziation zwischen $r_2.acv$ und $r_2.tv$
$r_2.ae_{tat}$	Objektvariablenassoziationsende an $r_2.tv$ von $r_2.ova_{at}$
$r_3.ova_{ca}$	Objektvariablenassoziation zwischen $r_3.c1v$ und $r_3.acv$
$r_3.ae_{cca}$	Objektvariablenassoziationsende an $r_3.cv$ von $r_3.ova_{ca}$
$r_3.ova_{at}$	Objektvariablenassoziation zwischen $r_3.acv$ und $r_3.tv$
$r_3.ae_{tat}$	Objektvariablenassoziationsende an $r_3.tv$ von $r_3.ova_{at}$
$r_3.ova_{ce0}$	Objektvariablenassoziation zwischen $r_3.c0v$ und $r_3.cae0v$
$r_3.ae_{cce0}$	Objektvariablenassoziationsende an $r_3.c0v$ von $r_3.ova_{ce0}$
$r_3.ova_{ce1}$	Objektvariablenassoziation zwischen $r_3.c1v$ und $r_3.cae1v$
$r_3.ae_{cce1}$	Objektvariablenassoziationsende an $r_3.c1v$ von $r_3.ova_{ce1}$
$r_3.ova_{ea0}$	Objektvariablenassoziation zwischen $r_3.cae0v$ und $r_3.cav$
$r_3.ae_{eea0}$	Objektvariablenassoziationsende an $r_3.cae0v$ von $r_3.ova_{ea0}$
$r_3.ae_{aea0}$	Objektvariablenassoziationsende an $r_3.cav$ von $r_3.ova_{ea0}$
$r_3.ova_{ea1}$	Objektvariablenassoziation zwischen $r_3.cae1v$ und $r_3.cav$
$r_3.ae_{eea1}$	Objektvariablenassoziationsende an $r_3.cae1v$ von $r_3.ova_{ea1}$
$r_3.ae_{aea1}$	Objektvariablenassoziationsende an $r_3.cav$ von $r_3.ova_{ea1}$

Tabelle B.2: Für den Nachweis der Metamodellkonformität benötigte Bezeichner (1/2)

Name	Bezeichnet
$r_4.ova_{at}$	Objektvariablenassoziation zwischen $r_4.acv$ und $r_4.tv$
$r_4.ae_{tat}$	Objektvariablenassoziationsende an $r_4.tv$ von $r_4.ova_{at}$
$r_4.ova_{ca}$	Objektvariablenassoziation zwischen $r_4.cv$ und $r_4.acv$
$r_4.ae_{cca}$	Objektvariablenassoziationsende an $r_4.cv$ von $r_4.ova_{ca}$
$r_4.ova_{ce0}$	Objektvariablenassoziation zwischen $r_4.c0v$ und $r_4.cae0v$
$r_4.ae_{cce0}$	Objektvariablenassoziationsende an $r_4.c0v$ von $r_4.ova_{ce0}$
$r_4.ova_{ce1}$	Objektvariablenassoziation zwischen $r_4.c1v$ und $r_4.cae1v$
$r_4.ae_{cce1}$	Objektvariablenassoziationsende an $r_4.c1v$ von $r_4.ova_{ce1}$
$r_4.ova_{ea0}$	Objektvariablenassoziation zwischen $r_4.cae0v$ und $r_4.cav$
$r_4.ae_{eea0}$	Objektvariablenassoziationsende an $r_4.cae0v$ von $r_4.ova_{ea0}$
$r_4.ae_{aea0}$	Objektvariablenassoziationsende an $r_4.cav$ von $r_4.ova_{ea0}$
$r_4.ova_{ea1}$	Objektvariablenassoziation zwischen $r_4.cae1v$ und $r_4.cav$
$r_4.ae_{eea1}$	Objektvariablenassoziationsende an $r_4.cae1v$ von $r_4.ova_{ea1}$
$r_4.ae_{aea1}$	Objektvariablenassoziationsende an $r_4.cav$ von $r_4.ova_{ea1}$
$r_5.ova_{ca}$	Objektvariablenassoziation „class-att“ zwischen $r_5.cv$ und $r_5.acv$
$r_5.ae_{cca}$	Objektvariablenassoziationsende an $r_5.cv$ von $r_5.ova_{ca}$
$r_5.ova_{at}$	Objektvariablenassoziation zwischen $r_5.acv$ und $r_5.tv$
$r_5.ae_{tat}$	Objektvariablenassoziationsende an $r_5.tv$ von $r_5.ova_{at}$
$r_5.ova_{cp}$	Objektvariablenassoziation „class-key“ zwischen $r_5.cv$ und $r_5.acv$
$r_5.ae_{ccp}$	Objektvariablenassoziationsende an $r_5.cv$ von $r_5.ova_{cp}$
$r_6.ova_{ce}$	Objektvariablenassoziation zwischen $r_6.cv$ und $r_6.caev$
$r_6.ae_{cce}$	Objektvariablenassoziationsende an $r_6.cv$ von $r_6.ova_{ce}$
$r_64.ova_{ea}$	Objektvariablenassoziation zwischen $r_6.caev$ und $r_6.cav$
$r_6.ae_{eea}$	Objektvariablenassoziationsende an $r_6.caev$ von $r_6.ova_{ea}$
$r_6.ae_{aea}$	Objektvariablenassoziationsende an $r_6.cav$ von $r_6.ova_{ea}$

Tabelle B.3: Für den Nachweis der Metamodellkonformität benötigte Bezeichner (2/2)

D.h. Forderung 4.2.10 (ii) ist erfüllt, Regel r_2 erzeugt eine Teilstruktur jedes Modellfragments das von Regel r_5 erzeugt wird. Folglich sind alle in der Superstruktur enthaltenen Objektvariablenassoziationen redundant.

Zu zeigen: $\neg relevant(r_5.ova_{ca}, r_5, R_{ER})$ gemäß Satz 4.2.11 (S. 138).

(i) $OVP(r_5, r_2) = \{\{(r_5.cv, r_2.cv), (r_5.acv, r_2.acv), (r_5.tv, r_2.tv)\}\}$

(ii) Betrachtet wird $r_2.ova^2_{ca}$:

$$
\begin{aligned}
& OVP_0 = \{(r_5.cv, r_2.cv), (r_5.acv, r_2.acv), (r_5.tv, r_2.tv)\} \in OVP(r_5, r_2) \\
& r_2.ova_{ca} = (r_5.ova_{ca}|_{OVAT}, 1, OVAE^2) \in r_5|_{mv_1}|_{OVA} \\
& \text{ mit } OVAE^2 := \{(ae_c, r2.cv), (ae_a, r2.av)\} \\
& \quad (ae_c, r_5.cv) \in r_5.ova_{ca} \\
& \quad (r_5.cv, r_2.cv) \in OVP_0 \\
& \quad (ae_a, r_5.av) \in r_5.ova_{ca} \\
& \quad (r_5.av, r_2.av) \in OVP_0 \\
& \quad r_2.ova_{ca}|_{cardv} = 1 \geq 1 = r_5.ova_{ca}|_{cardv}
\end{aligned}
$$

$$
\begin{aligned}
& \Rightarrow \quad redundant(R_{ER}, r_5, r_5.ova_{ca}) \\
& \overset{\text{Def. 4.2.12}}{\Leftrightarrow} \quad \neg relevant(R_{ER}, r_5, r_5.ova_{ca})
\end{aligned}
\tag{B.2}
$$

Zu zeigen: $\neg relevant(r_5.ova_{at}, r_5, R_{ER})$ gemäß Satz 4.2.11.

(i) $OVP(r_5, r_2) = \{\{(r_5.cv, r_2.cv), (r_5.acv, r_2.acv), (r_5.tv, r_2.tv)\}\}$

(ii) Betrachtet wird $r_2.ova_{at}$:

$$
\begin{aligned}
& OVP_0 = \{(r_5.cv, r_2.cv), (r_5.acv, r_2.acv), (r_5.tv, r_2.tv)\} \in OVP(r_5, r_2) \\
& r_2.ova_{at} = (r_5.ova_{at}|_{OVAT}, 1, OVAE^2) \in r_5|_{mv_1}|_{OVA} \\
& \text{ mit } OVAE^2 := \{(ae_a, r2.acv), (ae_t, r2.tv)\} \\
& \quad (ae_a, r_5.acv) \in r_5.ova_{at} \\
& \quad (r_5.acv, r_2.acv) \in OVP_0 \\
& \quad (ae_t, r_5.tv) \in r_5.ova_{at} \\
& \quad (r_5.tv, r_2.tv) \in OVP_0 \\
& \quad r_2.ova_{at}|_{cardv} = 1 \geq 1 = r_5.ova_{at}|_{cardv}
\end{aligned}
$$

$$
\begin{aligned}
& \Rightarrow \quad redundant(R_{ER}, r_5, r_5.ova_{at}) \\
& \overset{\text{Def. 4.2.12}}{\Leftrightarrow} \quad \neg relevant(R_{ER}, r_5, r_5.ova_{at})
\end{aligned}
\tag{B.3}
$$

Für alle anderen Paare aus Regeln existiert kein gültiges *OVP*, so dass alle weiteren Objektvariablenassoziationen als relevant angesehen werden müssen.

Objektvariable die identische Objekte erzeugen können

Gemäß Definition 4.2.17 (s. S. 145) erhält man für die Menge $\mathbb{OV}_{R_{ER}}^{conf*}$ mit den Mengen von Objektvariablen, welche potentiell dasselbe Zielobjekt erzeugen können, das folgende Ergebnis:

$$\begin{aligned}\mathbb{OV}_{R_{ER}}^{conf*} = \{\quad & \{r_1.cv, r_2.cv, r_3.c0v, r_3.c1v, r4.c0v, r4.c1v, r_5.cv, r_6.cv\},\\ & \{r_2.acv, r_5.acv\},\\ & \{r_3.acv\},\\ & \{r_4.acv\},\\ & \{r_2.tv, r_3.tv, r_4.tv, r_5.tv\},\\ & \{r_3.cae0v\},\\ & \{r_3.cae1v\},\\ & \{r_4.cae0v\},\\ & \{r_4.cae1v\},\\ & \{r_6.caev\},\\ & \{r_3.cav\},\\ & \{r_4.cav\},\\ & \{r_6.cav\}\ \}\end{aligned}$$

Abschätzung der Obergrenzen

Für die Abschätzung der Obergrenzenkonformität werden im weiteren nur die Klassenassziationsenden mit einer Multiplizitätsobergrenze kleiner als ∞ betrachtet. Dies sind im einzelnen die Klassenassoziationsenden CA_c, CK_c, AT_t, CE_c, EA_e und EA_a.

Klassenassoziationsende CA_c:
Betrachtet wird das Klassenassoziationsende CA_c mit $CA_c|_m = (1,1) =: (lb, ub)$:

Im Folgenden werden einige Werte für *ubVarCard* berechnet die für weiteren Beweis benötigt werden:

$$\begin{aligned}&dependsOn(r_2.cv) = \{r_2.ev\}\\ &dependsOn(r_2.acv) = \{r_2.aev\}\\ &\{r_2.aev\} \overset{r_2|_{mv0}}{\rightsquigarrow} \{r_2.ev\}\\ &\overset{\text{Tab. 4.2, Fall } (ix)}{\Rightarrow} ubVarCard(r_2.ova_{ca}, r_2.ae_{cca}, r_2) = 1\end{aligned}$$

$$\begin{aligned}&dependsOn(r_3.c1v) = \{r_3.ev, r_3.aev\}\\ &dependsOn(r_3.acv) = \{r_3.aev\}\\ &\{r_3.aev\} \overset{r_3|_{mv0}}{\rightsquigarrow} \{r_3.ev, r_3.aev\}\\ &\overset{\text{Tab. 4.2, Fall } (ix)}{\Rightarrow} ubVarCard(r_3.ova_{ca}, r_3.ae_{cca}, r_3) = 1\end{aligned}$$

$$dependsOn(r_4.c1v) = \{r_4.ev, r_4.aev\}$$
$$dependsOn(r_4.acv) = \{r_4.aev\}$$
$$\{r_4.aev\} \overset{r_4|_{mv_0}}{\rightsquigarrow} \{r_4.ev, r_4.aev\}$$
$$\overset{\text{Tab. 4.2, Fall } (ix)}{\Rightarrow} ubVarCard(r_4.ova_{ca}, r_4.ae_{cca}, r_4) = 1$$

$$\max_{OV_k \in \left\{ \begin{array}{l} \{(r_2,r_2.acv),(r_5,r_5.acv)\} \\ \{(r_3,r_3.acv)\} \\ \{(r_4,r_4.acv)\} \end{array} \right\}} \left\{ \sum_{\substack{r_l, ova_m: \\ \exists (r_l, ov_h) \in OV_k: \\ (oppositeEnd(AE_i, CA_c), ov_m) \in ova|_{OVAE} \\ \wedge ova_m|_{OVAT} = AE_i \\ \wedge relevant(R, r_l, ova_m)}} ubVarCard(ova_m, CA_c, r_l) \right\}$$
$$\overset{(B.2)^1}{=} max\left\{ \begin{array}{l} ubVarCard(r_2.ova_{ca}, r_2.ae_{cca}, r_2), \\ ubVarCard(r_3.ova_{ca}, r_3.ae_{cca}, r_3), \\ ubVarCard(r_4.ova_{ca}, r_4.ae_{cca}, r_4) \end{array} \right\}$$
$$= max\{1\} = 1 \leq 1 = ub$$

Klassenassoziationsende CK_c:
Betrachtet wird das Klassenassoziationsende CK_c mit $CK_c|_m = (0,1) =: (lb, ub)$:

Im Folgenden werden einige Werte für *ubVarCard* berechnet die für weiteren Beweis benötigt werden:

$$dependsOn(r_5.cv) = \{r_5.ev\}$$
$$dependsOn(r_5.acv) = \{r_5.aev\}$$
$$\{r_5.aev\} \overset{r_5|_{mv_0}}{\rightsquigarrow} \{r_5.aev\}$$
$$\overset{\text{Tab. 4.2, Fall } (ix)}{\Rightarrow} ubVarCard(r_5.ova_{cp}, r_5.ae_{ccp}, r_5) = 1$$

$$\max_{OV_k \in \left\{ \begin{array}{l} \{(r_2,r_2.acv),(r_5,r_5.acv)\} \\ \{(r_3,r_3.acv)\} \\ \{(r_4,r_4.acv)\} \end{array} \right\}} \left\{ \sum_{\substack{r_l, ova_m: \\ \exists (r_l, ov_h) \in OV_k: \\ (oppositeEnd(AE_i, CK_c), ov_m) \in ova|_{OVAE} \\ \wedge ova_m|_{OVAT} = AE_i \\ \wedge relevant(R, r_l, ova_m)}} ubVarCard(ova_m, CK_c, r_l) \right\}$$
$$= max\left\{ ubVarCard(r_5.ova_{ca}, r_5.ae_{cca}, r_5) \right\}$$
$$= max\{1\} = 1 \leq 1 = ub$$

Klassenassoziationsende AT_t:
Betrachtet wird das Klassenassoziationsende AT_t mit $AT_t|_m = (1,1) =: (lb, ub)$:

Im Folgenden werden einige Werte für *ubVarCard* berechnet die für weiteren Beweis benötigt

werden:

$$dependsOn(r_2.acv) = \{r_2.aev\}$$
$$dependsOn(r_2.tv) = \{r_2.wv\}$$
$$\{r_2.aev\} \overset{r_2|_{mv_0}}{\rightsquigarrow} \{r_2.wv\}$$
$$\overset{\text{Tab. 4.2, Fall } (ix)}{\Rightarrow} ubVarCard(r_2.ova_{at}, r_2.ae_{tat}, r_2) = 1$$

$$dependsOn(r_3.acv) = \{r_3.aev\}$$
$$dependsOn(r_3.tv) = \{r_3.wv\}$$
$$\{r_3.aev\} \overset{r_3|_{mv_0}}{\rightsquigarrow} \{r_3.wv\}$$
$$\overset{\text{Tab. 4.2, Fall } (ix)}{\Rightarrow} ubVarCard(r_3.ova_{at}, r_3.ae_{tat}, r_3) = 1$$

$$dependsOn(r_4.acv) = \{r_4.aev\}$$
$$dependsOn(r_4.tv) = \{r_4.wv\}$$
$$\{r_4.aev\} \overset{r_4|_{mv_0}}{\rightsquigarrow} \{r_4.wv\}$$
$$\overset{\text{Tab. 4.2, Fall } (ix)}{\Rightarrow} ubVarCard(r_4.ova_{at}, r_4.ae_{tat}, r_4) = 1$$

$$dependsOn(r_5.acv) = \{r_5.aev\}$$
$$dependsOn(r_5.tv) = \{r_5.wv\}$$
$$\{r_5.aev\} \overset{r_5|_{mv_0}}{\rightsquigarrow} \{r_5.wv\}$$
$$\overset{\text{Tab. 4.2, Fall } (ix)}{\Rightarrow} ubVarCard(r_5.ova_{at}, r_5.ae_{tat}, r_5) = 1$$

$$\max_{OV_k \in \left\{ \begin{array}{l} \{(r_2,r_2.acv),(r_5,r_5.acv)\} \\ \{(r_3,r_3.acv)\} \\ \{(r_4,r_4.acv)\} \end{array} \right\}} \left\{ \sum_{\substack{r_l, ova_m: \\ \exists (r_l, ov_h) \in OV_k: \\ (oppositeEnd(AT, AT_t), ov_m) \in ova|_{OVAE} \\ \wedge ova_m|_{OVAT} = AT \\ \wedge relevant(R, r_l, ova_m)}} ubVarCard(ova_m, AT_t, r_l) \right\}$$
$$\overset{(B.3)^2}{=} max\Big\{ ubVarCard(r_2.ova_{at}, r_2.ae_{tat}, r_2),$$
$$ubVarCard(r_3.ova_{at}, r_3.ae_{tat}, r_3),$$
$$ubVarCard(r_4.ova_{at}, r_4.ae_{tat}, r_4) \Big\}$$
$$= max\{1\} = 1 \leq 1 = ub$$

Klassenassoziationsende CE_c:
Betrachtet wird das Klassenassoziationsende CE_c mit $CE_c|_m = (1,1) =: (lb, ub)$:

Im Folgenden werden einige Werte für *ubVarCard* berechnet die für weiteren Beweis benötigt werden:

$$dependsOn(r_3.cae0v) = \diamondsuit$$
$$dependsOn(r_3.c0v) = \{r_3.ev\}$$
$$\overset{\text{Tab. 4.2, Fall } (xii)}{\Rightarrow} ubVarCard(r_3.ova_{ce0}, r_3.ae_{cce0}, r_3) = r_3.ova_{ce0}|_{cardv} = 1$$

$$dependsOn(r_3.cae1v) = \diamondsuit$$
$$dependsOn(r_3.c1v) = \{r_3.ev, r_3.aev\}$$
$$\overset{\text{Tab. 4.2, Fall } (xii)}{\Rightarrow} ubVarCard(r_3.ova_{ce1}, r_3.ae_{cce1}, r_3) = r_3.ova_{ce1}|_{cardv} = 1$$

$$dependsOn(r_4.cae0v) = \diamondsuit$$
$$dependsOn(r_4.c0v) = \{r_4.ev\}$$
$$\overset{\text{Tab. 4.2, Fall } (xii)}{\Rightarrow} ubVarCard(r_4.ova_{ce0}, r_4.ae_{cce0}, r_4) = r_4.ova_{ce0}|_{cardv} = 1$$

$$dependsOn(r_4.cae1v) = \diamondsuit$$
$$dependsOn(r_4.c1v) = \{r_4.ev, r_4.aev\}$$
$$\overset{\text{Tab. 4.2, Fall } (xii)}{\Rightarrow} ubVarCard(r_4.ova_{ce1}, r_4.ae_{cce1}, r_4) = r_4.ova_{ce1}|_{cardv} = 1$$

$$dependsOn(r_6.caev) = \{r_6.bev\}$$
$$dependsOn(r_6.cv) = \{r_6.ev\}$$
$$\{r_6.bev\} \overset{r_5|_{mv_0}}{\rightsquigarrow} \{r_6.ev\}$$
$$\overset{\text{Tab. 4.2, Fall } (ix)}{\Rightarrow} ubVarCard(r_6.ova_{ce}, r_6.ae_{cce}, r_6) = r_6.ova_{ce}|_{cardv} = 1$$

$$\max_{OV_k \in \left\{ \begin{array}{l} \{(r_3, r_3.cae0v)\}, \\ \{(r_3, r_3.cae1v)\}, \\ \{(r_4, r_4.cae0v)\}, \\ \{(r_4, r_4.cae1v)\}, \\ \{(r_6, r_6.caev)\} \end{array} \right\}} \left\{ \sum_{\substack{r_l, ova_m: \\ \exists (r_l, ov_h) \in OV_k: \\ (oppositeEnd(CE, CE_c), ov_m) \in ova|_{OVAE} \\ \wedge ova_m|_{OVAT} = CE \\ \wedge relevant(R, r_l, ova_m)}} ubVarCard(ova_m, CE_c, r_l) \right\}$$

$$= max\Big\{ ubVarCard(r_3.ova_{ce0}, r_3.ae_{cce0}, r_3),$$
$$ubVarCard(r_3.ova_{ce1}, r_3.ae_{cce1}, r_3),$$
$$ubVarCard(r_4.ova_{ce0}, r_4.ae_{cce0}, r_4),$$
$$ubVarCard(r_4.ova_{ce1}, r_4.ae_{cce1}, r_4) \qquad ubVarCard(r_6.ova_{ce}, r_6.ae_{cce}, r_6) \Big\}$$
$$= max\{1\} = 1 \leq 1 = ub$$

Klassenassoziationsende EA_e:
Betrachtet wird das Klassenassoziationsende EA_e mit $EA_e|_m = (2,2) =: (lb, ub)$:

Im Folgenden werden einige Werte für *ubVarCard* berechnet die für weiteren Beweis benötigt werden:

$$\begin{aligned}
&dependsOn(r_3.cav) = \diamondsuit \\
&dependsOn(r_3.cae0v) = \diamondsuit \\
&\overset{\text{Tab. 4.2, Fall } (xiii)}{\Rightarrow} ubVarCard(r_3.ova_{ea0}, r_3.ae_{eea0}, r_3) = r_3.ova_{ea0}|_{cardv} = 1
\end{aligned}$$

$$\begin{aligned}
&dependsOn(r_3.cav) = \diamondsuit \\
&dependsOn(r_3.cae1v) = \diamondsuit \\
&\overset{\text{Tab. 4.2, Fall } (xiii)}{\Rightarrow} ubVarCard(r_3.ova_{ea1}, r_3.ae_{eea1}, r_3) = r_3.ova_{ea1}|_{cardv} = 1
\end{aligned}$$

$$\begin{aligned}
&dependsOn(r_4.cav) = \diamondsuit \\
&dependsOn(r_4.cae0v) = \diamondsuit \\
&\overset{\text{Tab. 4.2, Fall } (xiii)}{\Rightarrow} ubVarCard(r_4.ova_{ea0}, r_4.ae_{eea0}, r_4) = r_4.ova_{ea0}|_{cardv} = 1
\end{aligned}$$

$$\begin{aligned}
&dependsOn(r_4.cav) = \diamondsuit \\
&dependsOn(r_4.cae1v) = \diamondsuit \\
&\overset{\text{Tab. 4.2, Fall } (xiii)}{\Rightarrow} ubVarCard(r_4.ova_{ea1}, r_4.ae_{eea1}, r_4) = r_4.ova_{ea1}|_{cardv} = 1
\end{aligned}$$

$$\begin{aligned}
&dependsOn(r_6.cav) = \{bv\} \\
&dependsOn(r_4.cae1v) = \{bev\} \\
&\neg(\{r_6.bv\} \overset{r_5|_{mv_0}}{\rightsquigarrow} \{r_6.bev\}) \\
&\overset{\text{Tab. 4.2, Fall } (xiii)}{\Rightarrow} ubVarCard(r_6.ova_{ea}, r_6.ae_{eea}, r_6) = \\
&\quad maxmatch(r_6|_{mv_0}, \{r_6.bv\}, \{r_6.bev\}) = 2
\end{aligned}$$

$$\begin{aligned}
&\max_{OV_k \in \left\{ \begin{array}{l} \{(r_3, r_3.cav)\}, \\ \{(r_4, r_4.cav)\}, \\ \{(r_6, r_6.cav)\} \end{array} \right\}} \left\{ \sum_{\substack{r_l, ova_m: \\ \exists (r_l, ov_h) \in OV_k: \\ (oppositeEnd(EA, EA_e), ov_m) \in ova|_{OVAE} \\ \wedge ova_m|_{OVAT} = EA \\ \wedge relevant(R, r_l, ova_m)}} ubVarCard(ova_m, EA_e, r_l) \right\} \\
&\overset{(B.3)^3}{=} max\Big\{ ubVarCard(r_3.ova_{ea0}, r_3.ae_{eea0}, r_3) + ubVarCard(r_3.ova_{ea1}, r_3.ae_{eea1}, r_3), \\
&\qquad ubVarCard(r_4.ova_{ea0}, r_4.ae_{eea0}, r_4) + ubVarCard(r_4.ova_{ea1}, r_4.ae_{eea1}, r_4), \\
&\qquad ubVarCard(r_6.ova_{ea}, r_6.ae_{eea}, r_6) \Big\} \\
&= max\{1\} = 2 \leq 2 = ub
\end{aligned}$$

Klassenassoziationsende EA_a:
Betrachtet wird das Klassenassoziationsende EA_a mit $EA_a|_m = (1,1) =: (lb, ub)$:

Im Folgenden werden einige Werte für *ubVarCard* berechnet die für weiteren Beweis benötigt werden:

$$dependsOn(r_3.cae0v) = \diamondsuit$$
$$dependsOn(r_3.cav) = \diamondsuit$$
$$\overset{\text{Tab. 4.2, Fall } (xiii)}{\Rightarrow} ubVarCard(r_3.ova_{ea0}, r_3.ae_{aea0}, r_3) = r_3.ova_{ea0}|_{cardv} = 1$$

$$dependsOn(r_3.cae1v) = \diamondsuit$$
$$dependsOn(r_3.cav) = \diamondsuit$$
$$\overset{\text{Tab. 4.2, Fall } (xiii)}{\Rightarrow} ubVarCard(r_3.ova_{ea1}, r_3.ae_{aea1}, r_3) = r_3.ova_{ea1}|_{cardv} = 1$$

$$dependsOn(r_4.cae0v) = \diamondsuit$$
$$dependsOn(r_4.cav) = \diamondsuit$$
$$\overset{\text{Tab. 4.2, Fall } (xiii)}{\Rightarrow} ubVarCard(r_4.ova_{ea0}, r_4.ae_{aea0}, r_4) = r_4.ova_{ea0}|_{cardv} = 1$$

$$dependsOn(r_4.cae1v) = \diamondsuit$$
$$dependsOn(r_4.cav) = \diamondsuit$$
$$\overset{\text{Tab. 4.2, Fall } (xiii)}{\Rightarrow} ubVarCard(r_4.ova_{ea1}, r_4.ae_{aea1}, r_4) = r_4.ova_{ea1}|_{cardv} = 1$$

$$dependsOn(r_4.cae1v) = \{bev\}$$
$$dependsOn(r_6.cav) = \{bv\}$$
$$\{r_6.bev\} \overset{r_5|_{mv0}}{\rightsquigarrow} \{r_6.bv\}$$
$$\overset{\text{Tab. 4.2, Fall } (ix)}{\Rightarrow} ubVarCard(r_6.ova_{ea}, r_6.ae_{eea}, r_6) = r_6.ova_{ea}|_{cardv} = 1$$

$$\max_{OV_k \in \left\{ \begin{array}{l} \{(r_3, r_3.cae0v)\}, \\ \{(r_3, r_3.cae1v)\}, \\ \{(r_4, r_4.cae0v)\}, \\ \{(r_4, r_4.cae1v)\}, \\ \{(r_6, r_6.caev)\} \end{array} \right\}} \left\{ \sum_{\substack{r_l, ova_m: \\ \exists (r_l, ov_h) \in OV_k: \\ (oppositeEnd(AE, AE_a), ov_m) \in ova|_{OVAE} \\ \wedge ova_m|_{OVAT} = AE \\ \wedge relevant(R, r_l, ova_m)}} ubVarCard(ova_m, AE_a, r_l) \right\}$$
$$= max\left\{ \begin{array}{l} ubVarCard(r_3.ova_{ea0}, r_3.ae_{aea0}, r_3), \\ ubVarCard(r_3.ova_{ea1}, r_3.ae_{aea1}, r_3), \\ ubVarCard(r_4.ova_{ea0}, r_4.ae_{aea0}, r_4), \\ ubVarCard(r_4.ova_{ea1}, r_4.ae_{aea1}, r_4) \qquad ubVarCard(r_6.ova_{ea}, r_6.ae_{eea}, r_6) \end{array} \right\}$$
$$= max\{1\} = 1 \leq 1 = ub$$

$$\overset{\text{(B.1), Satz 4.2.3}}{\Rightarrow} R_{ER} \text{ ist obergrenzenkonform} \tag{B.4}$$

B.3 Untergrenzenkonformität

Für die Abschätzung der Untergrenzenkonformität werden im weiteren nur die Klassenassziationsenden mit einer Multiplizitätsuntergrenze größer als 0 betrachtet. Dies sind im einzelnen die Klassenassoziationsenden CA_c, AT_t, CE_c, EA_e und EA_a.

Tabelle B.4 fasst die wesentlichen Schritte für den Nachweis der LB-Konformität der Regeln r_1 bis r_5 zusammen.

Es gilt zunächst:

$$\begin{aligned} dependsOn(r_6.cv) &= \{r_6.ev\} \\ dependsOn(r_6.caev) &= \{r_6.bev\} \\ dependsOn(r_6.cav) &= \{r_6.bv\} \end{aligned}$$

Für *lbCard* ergibt sich für die beiden relevanten Klassenassziationsenden:

$$\begin{aligned} & lbCard(CE, CE_c, r_6) \\ \overset{\text{Def. 4.2.29}}{=} & \min_{ova \in \{r_6.ova_{ce}\}} lbVarCard(ova, CE_c, r_6) \\ = & \; lbVarCard(r_6.ova_{ce}, CE_c, r_6) \\ \overset{\text{(B.5), (B.5), Tab. 4.3 Fall } (ix)}{=} & \; minmatch(r_6|_{mv_0}, \{r_6.bev\}, \{r_6.ev\}) \\ = & \; 1 \geq lb = 1 \end{aligned}$$

und

$$\begin{aligned} & lbCard(EA, EA_c, r_6) \\ \overset{\text{Def. 4.2.29}}{=} & \min_{ova \in \{r_6.ova_{ea}\}} lbVarCard(ova, EA_c, r_6) \\ = & \; lbVarCard(r_6.ova_{ea}, EA_c, r_6) \\ \overset{\text{(B.5), (B.5), Tab. 4.3 Fall } (ix)}{=} & \; minmatch(r_6|_{mv_0}, \{r_6.bv\}, \{r_6.ev\}) \\ = & \; 2 \geq lb = 2 \end{aligned}$$

$$\begin{aligned} & \overset{\text{Satz 4.2.5}}{\Rightarrow} && lbConform(r_6) \\ & \overset{\text{(B.1), Tab. B.4, Satz 4.2.6}}{\Rightarrow} && lbConform(R_{ER}) \\ & \overset{\text{(B.4), (B.5), Satz 4.2.7}}{\Longrightarrow} && R_{ER} \text{ ist metamodellkonform.} \end{aligned} \tag{B.5}$$

Regel	Assoziationsende	$lbConform$?
r_2	CA_c	$r_2.ova_{ca}\|_{cardv} = 1 \geq lb = 1$
r_2	AT_t	$r_2.ova_{at}\|_{cardv} = 1 \geq lb = 1$
		$\Rightarrow varLbConform(r_2)\|_{mv_1}$
		$\overset{\text{Satz 4.2.4}}{\Rightarrow} lbConform(r_2)$
r_3	CA_c	$r_3.ova_{ca}\|_{cardv} = 1 \geq lb = 1$
r_3	AT_t	$r_3.ova_{at}\|_{cardv} = 1 \geq lb = 1$
r_3	CE_c	$r_3.ova_{ce0}\|_{cardv} = 1 \geq lb = 1$
r_3	CE_c	$r_3.ova_{ce1}\|_{cardv} = 1 \geq lb = 1$
r_3	EA_e	$r_3.ova_{ea0}\|_{cardv}+$
		$r_3.ova_{ea0}\|_{cardv} = 2 \geq lb = 2$
r_3	EA_a	$r_3.ova_{ea0}\|_{cardv} = 1 \geq lb = 1$
r_3	EA_a	$r_3.ova_{ea1}\|_{cardv} = 1 \geq lb = 1$
		$\Rightarrow varLbConform(r_3)\|_{mv_1}$
		$\overset{\text{Satz 4.2.4}}{\Rightarrow} lbConform(r_3)$
r_4	CA_c	$r_4.ova_{ca}\|_{cardv} = 1 \geq lb = 1$
r_4	AT_t	$r_4.ova_{at}\|_{cardv} = 1 \geq lb = 1$
r_4	CE_c	$r_4.ova_{ce0}\|_{cardv} = 1 \geq lb = 1$
r_4	CE_c	$r_4.ova_{ce1}\|_{cardv} = 1 \geq lb = 1$
r_4	EA_e	$r_4.ova_{ea0}\|_{cardv}+$
		$r_4.ova_{ea0}\|_{cardv} = 2 \geq lb = 2$
r_4	EA_a	$r_4.ova_{ea0}\|_{cardv} = 1 \geq lb = 1$
r_4	EA_a	$r_4.ova_{ea1}\|_{cardv} = 1 \geq lb = 1$
		$\Rightarrow varLbConform(r_4)\|_{mv_1}$
		$\overset{\text{Satz 4.2.4}}{\Rightarrow} lbConform(r_4)$
r_5	CA_c	$r_5.ova_{ca}\|_{cardv} = 1 \geq lb = 1$
r_5	AT_t	$r_5.ova_{at}\|_{cardv} = 1 \geq lb = 1$
		$\Rightarrow varLbConform(r_5)\|_{mv_1}$
		$\overset{\text{Satz 4.2.4}}{\Rightarrow} lbConform(r_5)$
r_6	CE_c	$r_6.ova_{ce}\|_{cardv} = 1 \geq lb = 1$
r_6	EA_c	$r_6.ova_{ea}\|_{cardv} = 1 < lb = 1$
		$\Rightarrow$ Nachweis mit Hilfe von Satz 4.2.5

Tabelle B.4: Nachweis der Untergrenzenkonformität der einzelnen Regeln

C Das Regelwerk R_{AD}

Im Folgenden werden die Regeln des Regelwerks R_{AD} zur Abbildung von UML-Aktivitätsdiagrammen in ein konzeptuelles KOGITO-Modell vorgestellt. Die hier angeführten Regeln beschränken sich hierbei auf die, für das Beispiel aus Abschnitt 5.3 relevanten Aspekte. So werden durch das Regelwerk lediglich Aktivitäten vom Typ „Business Transaction Activity“ berücksichtigt. Die Berücksichtigung anderer Arten von KOGITO-Aktivitäten erfolgt jedoch vollkommen analog zu den hier vorgestellten Regeln.

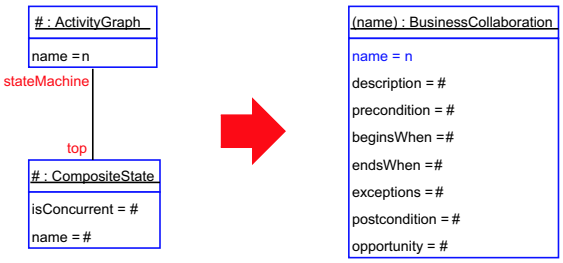


Abbildung C.1: Abbildungsregel r_{AD1}

Innerhalb der UML wird für jedes Aktivitätsdiagramm ein Objekt der Klasse ActivityGraph erzeugt. Dieses hat eine Referenz auf einen CompositeState, ein Container-Objekt für alle Zustände des Graphen, und Referenzen auf alle Transitionen des Diagramms. Die in Abbildung C.1 dargestellte Regel r_{AD1} erzeugt aus einem UML-Aktivitätsdiagramm eine KOGITO-Business-Collaboration gleichen Namens. Für Aktivitätsdiagramme ohne ein CompositeState-Objekt wird keine Business-Collaboration erzeugt, da diese offensichtlich leer sind.

Regel r_{AD2} aus Abbildung C.2 erzeugt zu einem Startzustand und einer Transition zu einem Objekt des Typs ActionState in einem UML-Aktiviätsdiagramm eine entsprechende Transition in der jeweiligen KOGITO-Business-Collaboration mit den jeweiligen Zuständen. Damit der ActionState als KOGTIO „Business Transaction Activity“ erkannt wird muss dieser, ebenso wie in allen anderen Regeln auch, durch einen Stereotyp mit dem entsprechenden Namen gekennzeichnet sein. Durch die Angabe des UML-Aktivitätsdiagramms und der zugehörigen KOGITO-Business-Collaboration wird, wie bei allen folgenden Regeln auch, sichergestellt, dass die Aktivitäten und Transitionen jeweils in der richtigen Business Collaboration angelegt werden.

Analog zu Regel r_{AD2} erzeugt *RAD* aus Abbildung C.3 eine Transition von einem Zustand zu einem Endzustand.

Die in Abbildung C.4 dargestellte Regel r_{AD4} bildet eine Transition zwischen zwei UML-Aktivitäten mit dem Stereotypen „Business Transaction Activity“ auf eine Transition und zwei entsprechende Aktivitäten im KOGITO-Modell ab.

Abbildung C.5 zeigt die Regel r_{AD5}. Sie bildet eine Guard-Bedingung aus dem UML-Modell auf eine KOGITO-Transition ab. Durch die Identität der Transition ist bereits sichergestellt, dass die Bedingung innerhalb der richten Business Collaboration und an der richtigen Transition eingefügt wird.

Analog zu Regel r_{AD5} bildet die in Abbildung C.6 dargestellte Regel r_{AD6} die Trigger-Bedingung

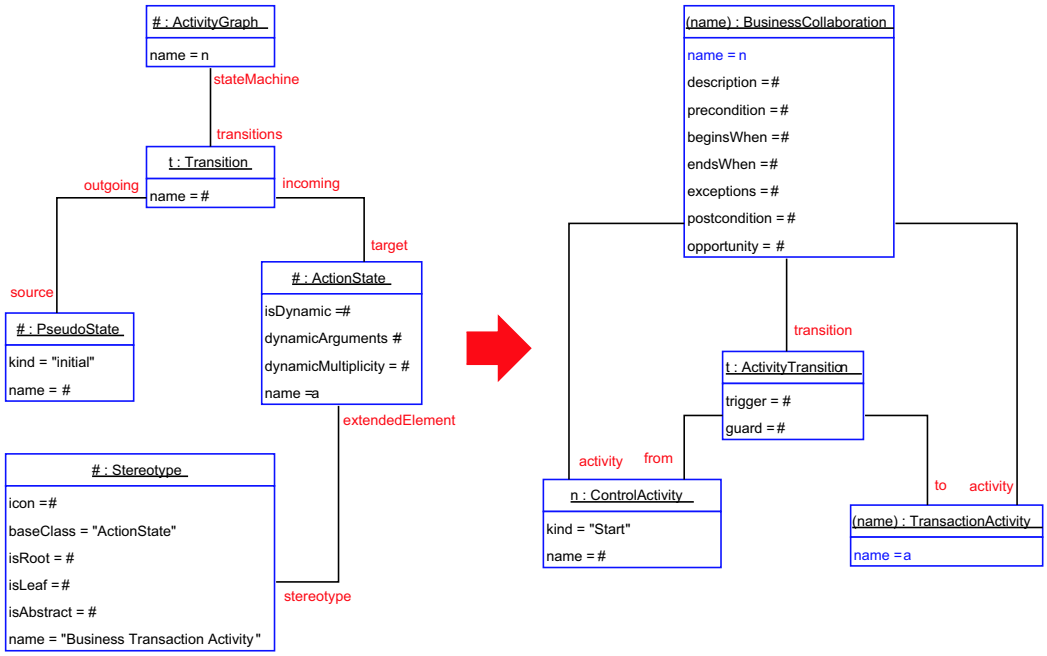


Abbildung C.2: Abbildungsregel r_{AD2}

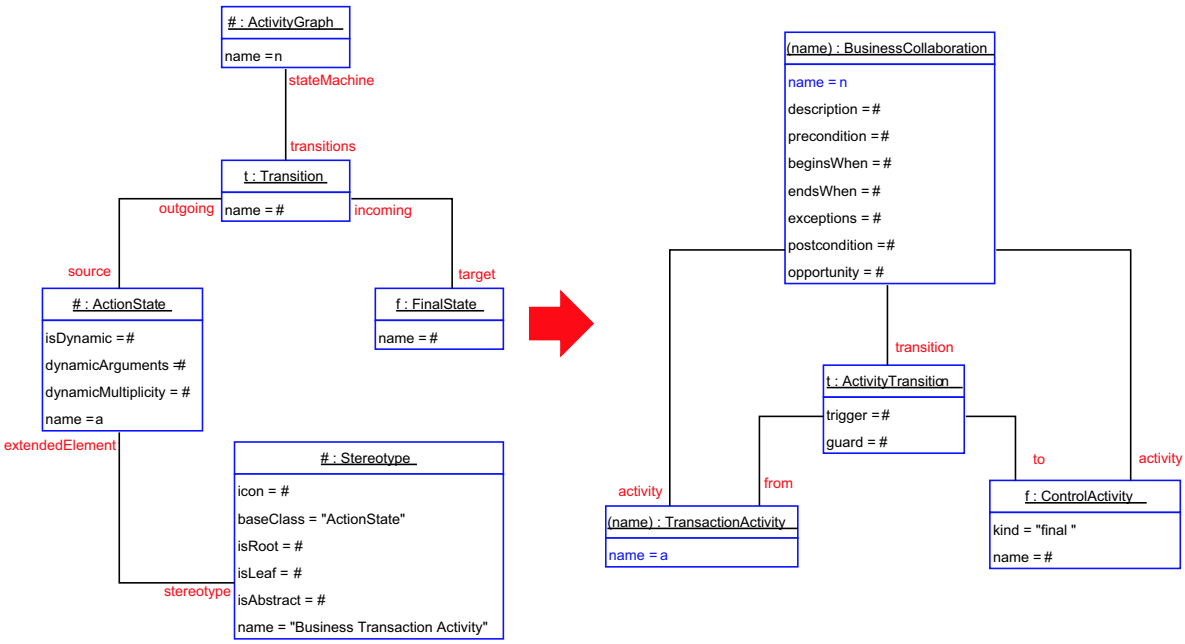


Abbildung C.3: Abbildungsregel r_{AD3}

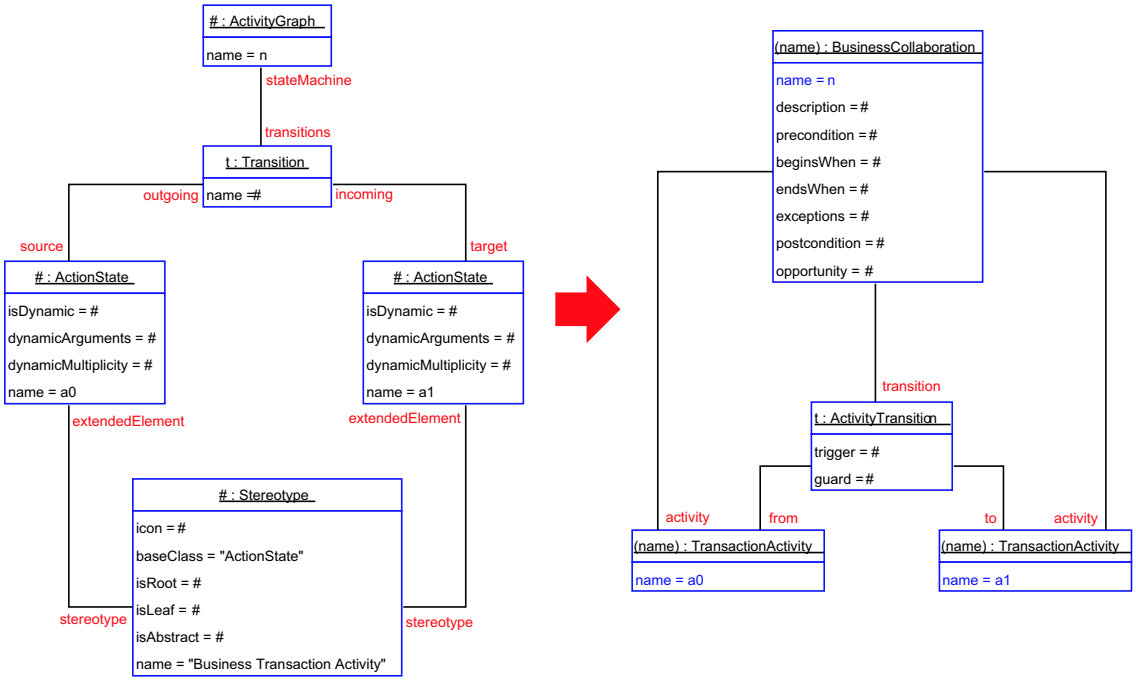


Abbildung C.4: Abbildungsregel r_{AD4}

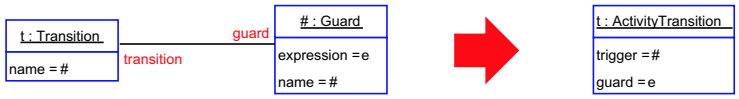


Abbildung C.5: Abbildungsregel r_{AD5}

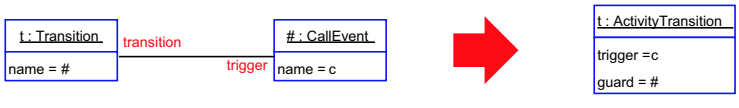


Abbildung C.6: Abbildungsregel r_{AD6}

einer Transition in ein KOGITO-Modell ab.

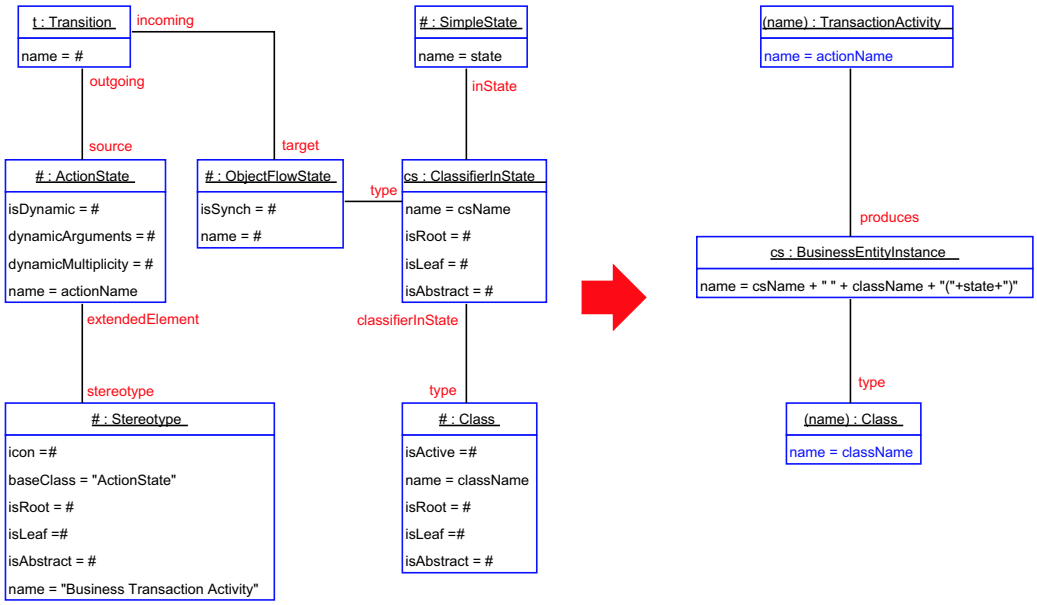


Abbildung C.7: Abbildungsregel r_{AD7}

Die Regeln r_{AD7} und r_{AD8} aus den Abbildungen C.7 und C.8 bilden Objektflusstransitionen von UML-Aktivitätsdiagrammen auf das KOGITO-Modell ab. Für jedes erzeugte Objekt wird eine Business Entity Instance im Kogito Modell erzeugt, die mit einer Klasse des KOGITO-Datenmodells assoziiert ist. Eine solche Klasse kann auch durch das Regelwerk R_{ER} erzeugt werden. Da das einzige Attribut name von Objekten des Typs Class ein Primärschlüsselattribut ist, wird auf ein evtl. bestehendes Class-Objekt verwiesen, falls ein solches bereits erzeugt wurde.

Befindet sich eine Business Transaction Activity innerhalb einer Swimlane, so ist der entsprechende UML-ActionState Teil einer gleichnamigen Partition. Regel r_{AD9} aus Abbildung C.9 legt für die Partition einen Aktor des Typs System Component an, der die Transaktion initialisiert.

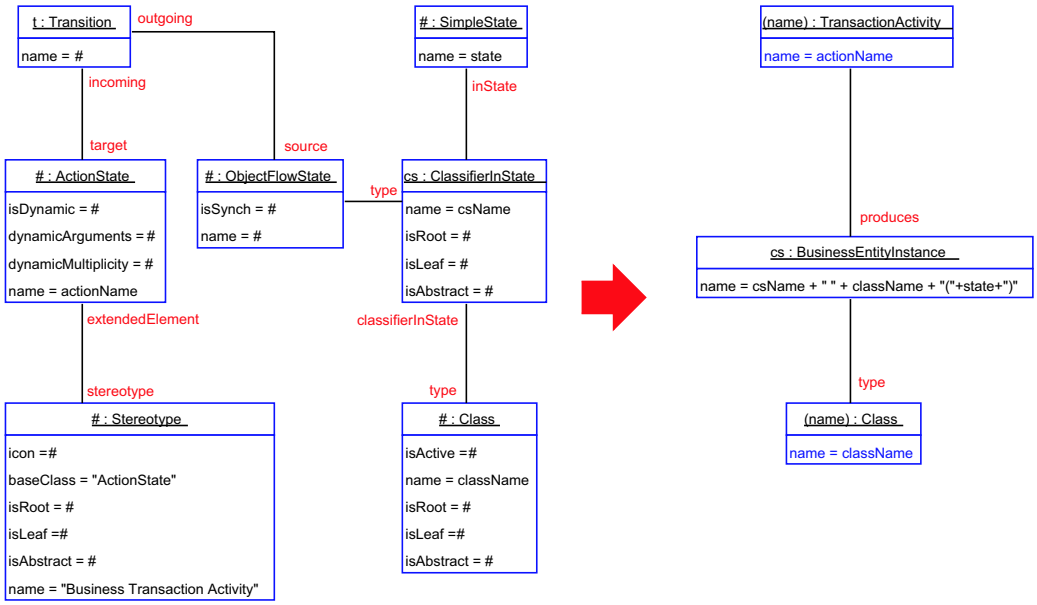


Abbildung C.8: Abbildungsregel r_{AD8}

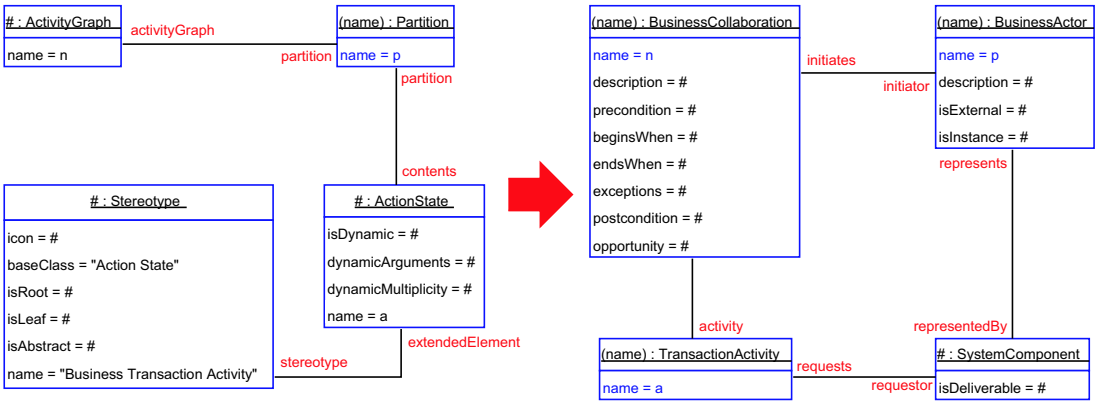


Abbildung C.9: Abbildungsregel r_{AD9}

D Beispiel zu Abschnitt 5.3.2

Innerhalb dieses Abschnitts wird durch Angabe eines Gegenbeispiels nachgewiesen, dass die durch das Regelwerk $R_{BRM} \cup R_{ER} \cup R_{AD}$ spezifizierte Integrationstransformation keine Verfeinerungskompostion bezüglich der Abstraktion R_a ist. Die verschiedenen Regelwerke werden in Abschnitten 5.2 und 5.3 eingeführt.

Im folgenden seien die Modelle $m'_{BRM} = (\emptyset, \emptyset)$ und $m'_{ER} = (\emptyset, \emptyset)$ leer. Das Modell m'_{AD} besteht aus einem Aktivitätsdiagramm mit dem Namen „Ghost Process“, welches eine einzige Business Transaction Aktiviät „Ghost Activity“ innerhalb einer Swimlane mit dem Namen „Ghost Component“ enthält. Abbildung D.1 stellt das Modell der abstrakten Syntax hierfür dar.

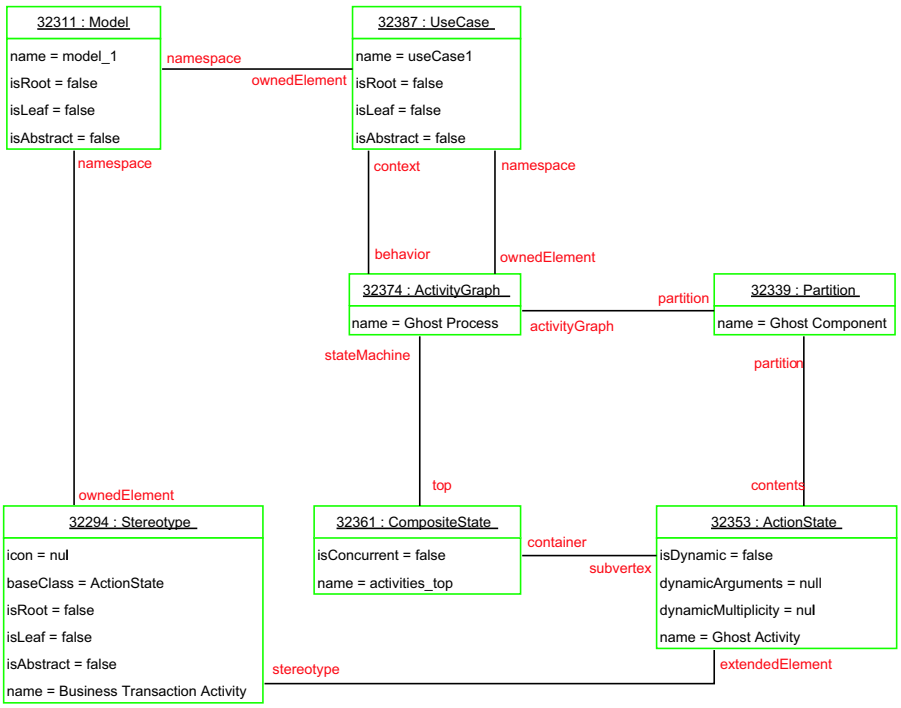


Abbildung D.1: Das Modell m'_{AD}

Bei der Anwendung des Regelwerks $R_{BRM} \cup R_{ER} \cup R_{AD}$ mit diesen Quellmodellen erzeugt offenbar lediglich die Regel r_{AD9} (siehe Abbildung C.9, S. 339) ein Zielmodellfragment. Das Ergebnis m'_{RAM} der Anwendung des Regelwerks ist in Abbildung D.2 dargestellt. Die Werte false in Attributen Boolschen Typs sowie die leeren Zeichenketten in String-Attributen entstehen durch den Eintrag von Default-Werten in nicht beschriebene Attributwerte am Ende der Transformation.

Wird auf dieses Ergebnis nun wiederum die durch das Regelwerk R_a spezifizierte Abstraktionsabbildung angewendet, so erhält man das in Abbildung dargestellte D.3 Modell m''_{BRM}. Offensichtlich ist dieses Modell weder identisch noch isomorph zu dem leeren Modell m'_{BRM}. Demzufolge stellt die durch das Regelwerk $R_{BRM} \cup R_{ER} \cup R_{AD}$ spezifizierte Abbildung keine Verfeinerungskomposition

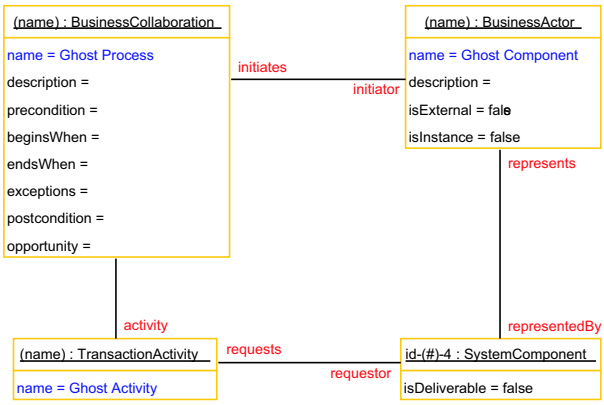


Abbildung D.2: Ergebnis der Transformation $m'_{RAM} := transform(\{m'_{AD}, (\emptyset, \emptyset)\}, R_{BRM} \cup R_{ER} \cup R_{AD})$

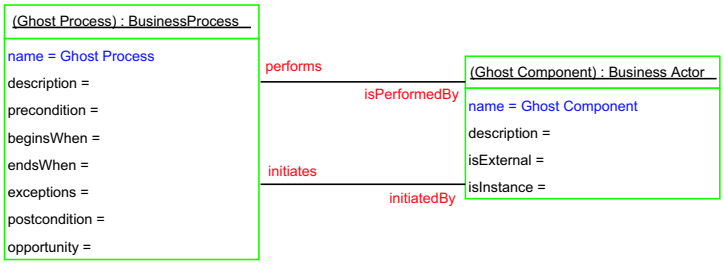


Abbildung D.3: Ergebnis der Transformation $m''_{BRM} := transform\big(transform(m'_{RAM}, R_a)$

bezüglich der Abstraktion R_a gemäß Definition 2.2.10 (siehe S. 46) dar.

E Automatisch generierter Nachweis der Metamodellkonformität

Die Abbildungen E.1 und E.2 stellen die beiden Regeln des in Abschnitt 3.3 vorgestellten Regelwerks *R* dar. Die Regeln wurden mit Hilfe des in Kapitel 6 vorgestellten Werkzeuges spezifiziert. Die einzelnen Objektvariablen wurden durch das Werkzeug automatisch mit Namen beschriftet. Im Anschluss ist der durch die Werkzeugunterstützung automatisch erstellte Nachweis der Metamodellkonformität des Regelwerks *R* angeführt.

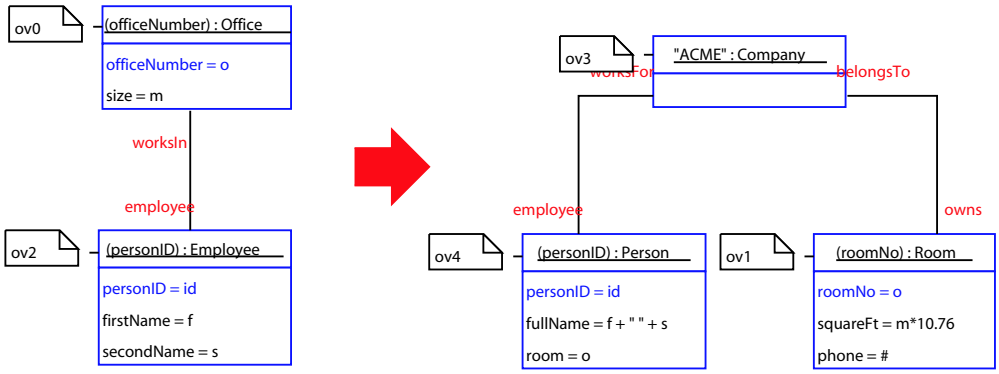


Abbildung E.1: Die Regel `r1`

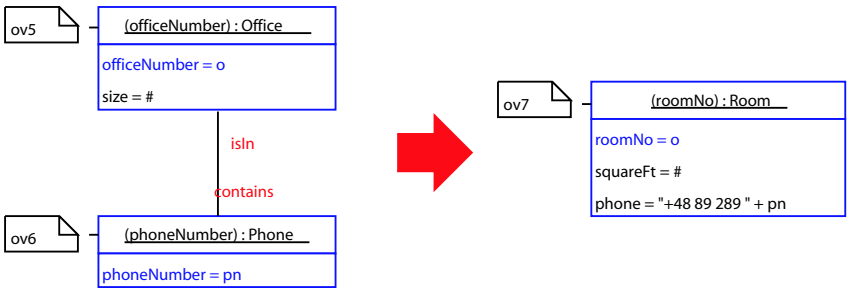


Abbildung E.2: Die Regel `r1`

```
Verification of the applicability of rule set R:

  Verification of the applicability of every rule in R:

  Verification of the applicability of rule r0

    Verification of the syntax of rule r0:

      OK - There are no expressions in the identifiers of target object variables.
      OK - There are no # values in primary key attribute variables of of target object variables.
      OK - There are no tuple values in attribute variables of of target object variables.
      OK - There are no tuple values in attribute variables of of source object variables.
      OK - There are string expressions in attribute variables of of source object variables.
      OK - No variable name starts with "botl".
```

```
  Rule r0 is syntactically correct.

  Verification of CreatesValidFragments(r0) according to Theorem 4.1.2

    Verification of Theorem 4.1.2 (i): The equation system GL must not have more than one solution.
    Equational system GL for rule r0
    GL_0^id:
                                    ov0|oiv = pK(ov0.officeNumber)
                                    ov2|oiv = pK(ov2.personID)
    GL_0^att:
                            ov0.officeNumber = o
                                   ov0.size = m
                                ov2.personID = id
                               ov2.firstName = f
                              ov2.secondName = s
    GL_1^id:
                              pK(ov1.roomNo) = ov1|oiv
                                       "ACME" = ov3|oiv
                            pK(ov4.personID) = ov4|oiv
    GL_1^att:
                                          o = ov1.roomNo
                                  m*10.76 = ov1.squareFt
                                          # = ov1.phone
                                       id = ov4.personID
                                f + " " + s = ov4.fullName
                                          o = ov4.room
    This rule is not constrainted.
    The solution of the equational system is:

                                    ov1|oiv = pK( (roomNo, ov0.officeNumber) )
                                  ov1.roomNo = ov0.officeNumber
                                ov1.squareFt = (ov0.size/25)*269
                                   ov1.phone = #
                                    ov3|oiv = "ACME"
                                    ov4|oiv = pK( (personID, ov2.personID) )
                                ov4.personID = ov2.personID
                                ov4.fullName = ov2.firstName +  " " + ov2.secondName
                                    ov4.room = ov0.officeNumber

    OK Theorem 4.1.2(i) is fulfilled

    Verification of Theorem 4.1.2 (ii): No two object variables of the target model variable in r0 should be similar.
      -  Verification of ov3[Company] ~ ov1[Room]:
         ov3[Company] ~ ov1[Room] = false (The object  variables are of different types (Definition 3.5.8 (i)))
      OK - ov1[Room] !~ ov3[Company]
      -  Verification of ov4[Person] ~ ov1[Room]:
         ov4[Person] ~ ov1[Room] = false (The object  variables are of different types (Definition 3.5.8 (i)))
      OK - ov1[Room] !~ ov4[Person]
      -  Verification of ov4[Person] ~ ov3[Company]:
         ov4[Person] ~ ov3[Company] = false (The object  variables are of different types (Definition 3.5.8 (i)))
      OK - ov3[Company] !~ ov4[Person]
    OK Theorem 4.1.2(i) is fulfilled

  OK - createsValidFragments(r0) = true.

  Verification if all target object variables in rule r0 are deterministic:

    deterministic(r0, ov1[Room]): Theorem 4.1.4 (4.15) required for the proof.
       dependsOn(ov1[Room], r0) = {ov0[Office]}
       attDependsOn(ov1[Room].squareFt[ = m*10.76]) = {ov0[Office]}
       {ov0[Office]} --determines(r0)--> {ov0[Office]} = true
       attDependsOn(ov1[Room].phone[ = #]) = {} (empty set)
       {ov0[Office]} --determines(r0)--> {} = true (second set is empty)
    OK deterministic(ov1[Room]) = true
    OK deterministic(r0, ov3[Company]) = true (Theorem 4.1.4 (4.14): all attribute variables are constant or #)
    deterministic(r0, ov4[Person]): Theorem 4.1.4 (4.15) required for the proof.
       dependsOn(ov4[Person], r0) = {ov2[Employee]}
       attDependsOn(ov4[Person].fullName[ = f + " " + s]) = {ov2[Employee]}
       {ov2[Employee]} --determines(r0)--> {ov2[Employee]} = true
       attDependsOn(ov4[Person].room[ = o]) = {ov0[Office]}
       {ov2[Employee]} --determines(r0)--> {ov0[Office]} = true
    OK deterministic(ov4[Person]) = true

  Rule r0 is deterministic. All object variables in this rule are deterministic.

  Verification if rule r0 is conflictFree.
  OK Rule r0 is conflictFree.

Rule r0 is applicable.

Verification of the applicability of rule r1

  Verification of the syntax of rule r1:

    OK - There are no expressions in the identifiers of target object variables.
    OK - There are no # values in primary key attribute variables of of target object variables.
    OK - There are no tuple values in attribute variables of of target object variables.
    OK - There are no tuple values in attribute variables of of source object variables.
    OK - There are string expressions in attribute variables of of source object variables.
    OK - No variable name starts with "botl".
```

```
    Rule r1 is syntactically correct.

    Verification of CreatesValidFragments(r1) according to Theorem 4.1.2

      Verification of Theorem 4.1.2 (i): The equation system GL must not have more than one solution.
      Equational system GL for rule r1
      GL_0^id:
                                      ov5|oiv = pK(ov5.officeNumber)
                                      ov6|oiv = pK(ov6.phoneNumber)
      GL_0^att:
                              ov5.officeNumber = o
                                     ov5.size = #                          (ignored according to Definition 3.5.4)
                               ov6.phoneNumber = pn
      GL_1^id:
                                pK(ov7.roomNo) = ov7|oiv
      GL_1^att:
                                            o = ov7.roomNo
                                            # = ov7.squareFt
                           "+48 89 289 " + pn = ov7.phone
      This rule is not constrainted.
      The solution of the equational system is:

                                      ov7|oiv = pK( (roomNo, ov5.officeNumber) )
                                    ov7.roomNo = ov5.officeNumber
                                  ov7.squareFt = #
                                     ov7.phone =  "+48 89 289 " + ov6.phoneNumber

      OK Theorem 4.1.2(i) is fulfilled

      Verification of Theorem 4.1.2 (ii): No two object variables of the target model variable in r1 should be similar.
      OK Theorem 4.1.2(i) is fulfilled

    OK - createsValidFragments(r1) = true.

    Verification if all target object variables in rule r1 are deterministic:

      deterministic(r1, ov7[Room]): Theorem 4.1.4 (4.15) required for the proof.
         dependsOn(ov7[Room], r1) = {ov5[Office]}
         attDependsOn(ov7[Room].squareFt[ = #]) = {} (empty set)
         {ov5[Office]} --determines(r1)--> {} = true (second set is empty)
         attDependsOn(ov7[Room].phone[ = "+48 89 289 " + pn]) = {ov6[Phone]}
         {ov5[Office]} --determines(r1)--> {ov6[Phone]} = true
      OK deterministic(ov7[Room]) = true

    Rule r1 is deterministic. All object variables in this rule are deterministic.

    Verification if rule r1 is conflictFree.
    OK Rule r1 is conflictFree.

  Rule r1 is applicable.

  Verification if the rule set R is conflictFree:

    Verification of conflictFree(R, ov1[Room], ov7[Room]):
       ov1[Room]|oiv ~ ov7[Room]|oiv = false (The type Room of ov1[Room] and ov7[Room] has primary keys.)
       => ! potConflicting (Lemma 4.1.5)
    OK => conflictFree(R, ov1[Room], ov7[Room]) (Lemma 4.1.6 (4.19))

  OK - Rule set R is conflict free.

Verification of the applicability according to Theorem 4.1.7: OK -
All rules of R are applicable (Theorem 4.1.7 (i)). OK - The rule
set is conflictFree (Theorem 4.1.7 (ii)). => OK R is applicable.

Verification of the upper bounds confromance of R according to
Theorem 4.2.3:

  Calculation of OV_(R)^(conf*):
    OV_(R)^(conf*) = { {ov1[Room], ov7[Room]}
                       {ov3[Company]}
                       {ov4[Person]} }

  Calculation of relevant target object variable associations:

    ov3[Company]-belongsTo--(1)--belongsTo-ov1[Room] in rule r0: Verification of relevance:
      redundant(R, r0, ov3[Company]-belongsTo--(1)--belongsTo-ov1[Room]) = false
    relevant(R, r0, ov3[Company]-belongsTo--(1)--belongsTo-ov1[Room]) = true

    ov3[Company]-worksFor--(1)--worksFor-ov4[Person] in rule r0: Verification of relevance:
      redundant(R, r0, ov3[Company]-worksFor--(1)--worksFor-ov4[Person]) = false
    relevant(R, r0, ov3[Company]-worksFor--(1)--worksFor-ov4[Person]) = true

  Calculation of the maximum number of outgoing associations for every class association end:

    End Company-belongsTo-(0, 1) of the class association Company-belongsTo-(0, 1)--(0, *)-owns-Room
    Estimation of the maximal possible number of outgoing association ends this type:

      Actual OVconf* entry = {ov1[Room], ov7[Room]}

        Calculation of ubVarCard(ov3[Company]-belongsTo--(1)--belongsTo-ov1[Room], Company-belongsTo-(0, 1), r0):
```

```
          dependsOn(ov1[Room], r0) = {ov0[Office]}
          dependsOn(ov3[Company], r0) = {} (ov3[Company]|oiv is constant)
        ubVarCard(ov3[Company]-belongsTo--(1)--belongsTo-ov1[Room], Company-belongsTo-(0, 1), r0) = ova|cardv = 1
        (Definition 4.2.21 Case 2 (iii))
      Sum of all OVconf* entry's ubVarCard values is 1.
    OK Company-belongsTo-(0, 1)|ub = 1 >= 1 = Estimated value

    End Room-owns-(0, *) of the class association Company-belongsTo-(0, 1)--(0, *)-owns-Room has an infinite upper bound.
    OK - No further verification necessary.

    End Company-worksFor-(1, 1) of the class association Company-worksFor-(1, 1)--(0, *)-employee-Person
    Estimation of the maximal possible number of outgoing association ends this type:

      Actual OVconf* entry = {ov4[Person]}

        Calculation of ubVarCard(ov3[Company]-worksFor--(1)--worksFor-ov4[Person], Company-worksFor-(1, 1), r0):
          dependsOn(ov4[Person], r0) = {ov2[Employee]}
          dependsOn(ov3[Company], r0) = {} (ov3[Company]|oiv is constant)
        ubVarCard(ov3[Company]-worksFor--(1)--worksFor-ov4[Person], Company-worksFor-(1, 1), r0) = ova|cardv = 1
        (Definition 4.2.21 Case 2 (iii))
      Sum of all OVconf* entry's ubVarCard values is 1.
    OK Company-worksFor-(1, 1)|ub = 1 >= 1 = Estimated value

    End Person-employee-(0, *) of the class association Company-worksFor-(1, 1)--(0, *)-employee-Person
    has an infinite upper bound.
    OK - No further verification necessary.
  All association ends are upper bounds conform.

OK Rule set R is upper bounds conform according to Theorem 4.2.3.

Verification of the lower bounds conformance of rule set R:

  Verfication of lbConform(r0):
    Calculation of varLbConform(r0|mv1)
      Class association end Company-belongsTo-(0, 1) in Company-belongsTo-(0, 1)--(0, *)-owns-Room
          Company-belongsTo-(0, 1) in Company-belongsTo-(0, 1)--(0, *)-owns-Room has a lower bound of 0.
        OK The actual cardinality from object variable ov3[Company] is 1 =< 0.
      Class association end Company-worksFor-(1, 1) in Company-worksFor-(1, 1)--(0, *)-employee-Person
          Company-worksFor-(1, 1) in Company-worksFor-(1, 1)--(0, *)-employee-Person has a lower bound of 0.
        OK The actual cardinality from object variable ov3[Company] is 1 =< 0.
      Class association end Room-owns-(0, *) in Company-belongsTo-(0, 1)--(0, *)-owns-Room
          Room-owns-(0, *) in Company-belongsTo-(0, 1)--(0, *)-owns-Room has a lower bound of 0.
        OK The actual cardinality from object variable ov1[Room] is 1 =< 0.
      Class association end Person-employee-(0, *) in Company-worksFor-(1, 1)--(0, *)-employee-Person
          Person-employee-(0, *) in Company-worksFor-(1, 1)--(0, *)-employee-Person has a lower bound of 1.
        OK The actual cardinality from object variable ov4[Person] is 1 =< 1.
    varLbConform(r0|mv1) = true

  Verfication of lbConform(r1):
    Calculation of varLbConform(r1|mv1)
      Class association end Company-belongsTo-(0, 1) in Company-belongsTo-(0, 1)--(0, *)-owns-Room
        OK: Target model variable of r1 contains no object variables of this end's type
      Class association end Company-worksFor-(1, 1) in Company-worksFor-(1, 1)--(0, *)-employee-Person
        OK: Target model variable of r1 contains no object variables of this end's type
      Class association end Room-owns-(0, *) in Company-belongsTo-(0, 1)--(0, *)-owns-Room
          Room-owns-(0, *) in Company-belongsTo-(0, 1)--(0, *)-owns-Room has a lower bound of 0.
        OK The actual cardinality from object variable ov7[Room] is 0 =< 0.
      Class association end Person-employee-(0, *) in Company-worksFor-(1, 1)--(0, *)-employee-Person
        OK: Target model variable of r1 contains no object variables of this end's type
    varLbConform(r1|mv1) = true

=> OK Rule set R is lower bounds conform.

Verification of metamodel conformance according to Theorem 4.2.7:
  OK Rule set R is applicable.
  OK lbConf(R)
  OK ubConf(R)
OK Rule set R is metamodel conform.
```

Literaturverzeichnis

[ABG+03] A. Alkassar, M. Broy, F. Gehring, M. Garschhammer, H.-G. Hegering, P. Keil, H. Kelter, U. Löwer, M. Pankow, A. Picot, A.-R. Sadeghi, M. Schiffers, M. Ullmann, and S. Vogel. *Kommunikations- und Informationstechnik 2010+3: Neue Trends und Entwicklungen in Technologie, Anwendungen und Sicherheit*. SecuMedia Verlag, Ingelheim, 2003.

[AG04] Gentleware AG. Poseidon for UML. Product Hompage: `http://www.gentleware.com/`, 2004.

[Agr04] Aditya Agrawal. GReAT - Graph-REwriting And Transformation language. Project Homepage: `http://aditya.isis.vanderbilt.edu/great.htm`, 2004.

[AHL00] K. Aardal, C. A. J. Hurkens, and A. K. Lenstra. Solving a system of linear diophantine equations with lower and upper bounds on the variables. *Mathematics of Operations Research*, 25(3):427–442, 2000.

[Ake00] David H Akehurst. *Model Translation: A UML-based specification technique and active implementation approach*. PhD thesis, University of Kent at Canterbury, 2000.

[AKS03] A. Agrawal, G. Karsai, and F. Shi. A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations. Technical report, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, 2003.

[AKSS03] A. Agrawal, G. Karsai, F. Shi, and J. Sprinkle. On the Use of Graph Transformations for the Formal Specification of Model Interpreters. *Journal of Universal Computer Science*, 9(11):1296–1322, November 2003.

[Aßm94] Uwe Aßmann. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *5th Int. Workshop on Graph Grammars and their Application To Computer Science, Williamsburg*, volume 1073 of *Lecture Notes in Computer Science*, pages 321–335, Heidelberg, November 1994. Springer.

[Aßm96] Uwe Aßmann. Graph Rewrite Systems For Program Optimization. Technical Report RR-2955, INRIA Rocquencourt, 1996.

[Aßm98a] Uwe Aßmann. A Tutorial for OPTIMIX, November 1998.

[Aßm98b] Uwe Aßmann. OPTIMIX Language Manual (for OPTIMIX 2.5), November 1998.

[Aßm99] Uwe Aßmann. *OPTIMIX, A Tool for Rewriting and Optimizing Programs*, volume 2, chapter 8, pages 307 – 318. Chapman-Hall, 1999.

[Amb98] Scott W. Ambler. *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press, Camebridge, UK, July 1998.

[Amr94] Beatrice Amrhein. Gröbner-Basen: Vorlesungsskript. Technical Report WSI-94-12, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Tübingen, 1994. ISSN 0946-3852.

[AP94] William W. Adams and Loustaunau Philippe. *An introduction to Grobner bases*, volume 3 of *Graduate Studies in Math*. Oxford University Press, 1994.

[BBE+03] Manfred Broy, Bernd Brügge, Jörg Eberspächer, Georg Färber, Gunther Reinhart, and Horst Wildemann. Bayerischer Forschungsverbund Software Engineering FORSOFT II - Abschlussbericht 2003, 2003.

[BD93] A.-P. Bröhl and W. Dröschel, editors. *Das V-Modell: Der Standard für die Softwareentwicklung mit Praxisleitfaden*. Software - Anwendungsentwicklung - Informationssysteme. R. Oldenburg Verlag, München, Wien, 1993.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley Professional, 1999.

[BEH04] Peter Braun, Florian Erhard, and Franz Huber. Die AUTOFOCUS Homepage. `http://autofocus.informatik.tu-muenchen.de/`, 2004.

[Bet04] John Bettin. GMT (Generative Model Transformer) Software Requirements Specification, Juli 2004. Version 0.1.

[BFG93] Sergio C. Bandinelli, Alfonso Fugetta, and Carlo Ghezzi. Software Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 1993.

[BM03a] Peter Braun and Frank Marschall. BOTL - The Bidirectional Object Oriented Transformation Language. Technischer Bericht TUM-I0307, Technische Universität München, Institut für Informatik, Mai 2003.

[BM03b] Peter Braun and Frank Marschall. Transforming Object Oriented Models with BOTL. In Paolo Bottoni and Mark Minas, editors, *International Workshop on Graph Transformation and Visual Modeling Techniques*, volume 72 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B. V., 2003.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns*. John Wiley & Sons Ltd, Chichester, England, 1996.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[BPPT03] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Coordinated distributed diagram transformation for software evolution. In Reiko Heckel, Tom Mens, and Michel Wermelinger, editors, *Electronic Notes in Theoretical Computer Science*, volume 72. Elsevier, 2003.

[BS01] Manfred Broy and Ketil Stolen. *Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement*. Monographs in Computer Science. Springer, Berlin, 2001.

[BW93] Thomas Becker and Volker Weispfenning. *Gröbner Bases. A Computational Approach to Commutative Algebra*, volume 141 of *Graduate Texts in Mathematics*. Springer, 1993.

[CEK+00] T. Clark, A.S. Evans, S. Kent, S. Brodsky, and S. Cook. *A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach*. 2000. Available at http://www.puml.org.

[CGK+02] Francisco Curbera, Yaron Goland, Johannes Klein, Frank Leymann, Dieter Roller, Satish Thatte, and Sanjiva Weerawarana. Business process execution language for web services, version 1.0. Initial Public Draft Release, July 2002.

[CGN99] K. Cremer, S. Gruner, and M. Nagl. *Handbook of Graph Grammars and Computing by Graph Tranformation*, volume 2 of *Applications Languages and Tools*, chapter 10. Graph Transformation-Based Integration Tools: Application To Chemical Process Engineering, pages 369–394. World Scientific, 1999.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Anaheim, October 2003.

[Cho56] Noam Chomsky. Three models for the description of language. *IRI Transactions on Information Theory*, 2(3):113–124, 1956.

[Cho59] Noam Chomsky. On Certain Formal Properties of Grammars]. *Information and Control*, 2:137–167, 1959.

[Chr02] Alexander Christoph. Graph Rewrite Systems for Software Design Transformations. In *Proceedings of the NET.ObjectDays*, 2002.

[C.J00] C.J.Clark. The UN/CEFACT Unified Modelling Methodology - An Overview. UNCEFACT/TMWG/N097, March 2000.

[CM03] Compuware Corporation and SUN Microsystems. XMOF: Queries, Views and Transformations on Models using MOF, OCL and Patterns. OMG Document ad/2003-03-24, March 2003. MOF 2.0 Query / Views / Transformations Initial Submission.

[Col04a] CollabNet. Argouml. Project Hompage: `http://argouml.tigris.org/`, 2004.

[Col04b] CollabNet. Java graph editing framework. Project Hompage: `http://gef.tigris.org/`, 2004.

[Com04] Compuware Corporation. Compuware OptimalJ Homepage. `http://www.compuware.com/products/optimalj/`, 2004.

[Cor00] Microsoft Corp. Biztalk (tm) framework 2.0: Document and message specification, December 2000.

[Cor04a] Borland Software Corporation. Together. Product Hompage: `http://www.borland.com/together/index.html`, 2004.

[Cor04b] IBM Corporation. Rational RequisitPro Homepage. `http://www-306.ibm.com/software/awdtools/reqpro/`, 2004.

[Cor04c] Microsoft Corporation. Microsoft office visio 2003. Product Hompage: `http://office.microsoft.com/assistance/topcategory.aspx?TopLevelCat=CH79001814&CTT=6&Origin=ES790020011031`, 2004.

[Doo04] Doors homepage. `http://www.telelogic.com/products/doorsers/doors/`, 2004.

[DW98] D. F. D'Souza and A. C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[eBPPT01] ebXML Business Process Project Team. ebxml business process specification schema v1.01. ebXML Technical Specification, May 2001.

[Ecl04] Eclipse Project Homepage. `http://www.eclipse.org`, 2004.

[EG04] ExoLab-Group. Castor. Project Hompage: `http://www.castor.org/`, 2004.

[EK99] A.S. Evans and S. Kent. *Core Meta-Modeling Semantics of UML: the pUML Approach*. 2nd International Conference on the Unified Modeling Language, 1999. Available at http://www.cs.york.ac.uk/puml.

[Eng93] Hermann Engesser, editor. *Duden „Informatik" : Ein Sachlexikon für Studium und Praxis / Hrsg. vom Lektorat d. BI-Wiss.-Verl. unter Leitung v. Hermann Engesser. Bearb. von Volker Claus u. Andreas Schwill*. Dudenverlag, Mannheim; Leipzig; Wien; Zürich, 1993.

[EP00] Hans-Erik Eriksson and Magnus Penker. *Business Modeling With UML: Business Patterns at Work*. OMG Press. John Wiley & Sons Inc, 2000.

[fEFTF01] OMG UML for EAI Finalization Task Force. UML Profile for Enterprise Application Integration (EAI). OMG Interim FTF Report ptc/2003-02-01, March 2001.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19:325–334, 1998.

[FGH+94] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multiperspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[FKN94] Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors. *Software process modelling and technology*. Research Studies Press Ltd., 1994.

[Fun04] Nathan Funk. JEP - Java Mathematical Expression Parser. Web page: `http://www.singularsys.com/jep/`, 2004.

[GGKH03] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. IBM Zurich Research Laboratory, e-Business Solutions, July 2003.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[GLR+02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The Missing Link of MDA. In Paolo Bottoni and Mark Minas, editors, *International Workshop on Graph Transformation and Visual Modeling Techniques*, number 72.3 in ENTCS. Elsevier Science B. V., 2002.

[GMP+01a] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, and Wolfgang Schwerin. Modular Process Patterns supporting an Evolutionary Software Development Process. In *Product Focused Software Process Improvement : Third International Conference, PROFES 2001*, volume 2188 / 2001 of *Lecture Notes in Computer Science*, page 326 ff. Springer-Verlag Heidelberg, Januar 2001.

[GMP+01b] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, and Wolfgang Schwerin. Towards a Living Software Development Process based on Process Patterns. In *Software Process Technology: 8th European Workshop, EWSPT 2001*, volume 2077 / 2001 of *Lecture Notes in Computer Science*, page 182 ff. Springer-Verlag Heidelberg, Januar 2001.

[GMP+02a] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, and Wolfgang Schwerin, editors. *Proceedings of the 1st Workshop on Software Development Process Patterns (SDPP'02)*, number TUM-I0213. Technische Universität München, Institut für Informatik, 2002.

[GMP+02b] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, and Wolfgang Schwerin. Towards a Tool Support for a Living Software Development Process. In *Proceedings of the Hawai'i International Conference on System Sciences*, 2002.

[GMP+03a] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, and Wolfgang Schwerin. LiSa - Living Software Development Process Support Tool. Project Homepage: `http://processpatterns.informatik.tu-muenchen.de/`, 2003.

[GMP+03b] Michael Gnatz, Frank Marschall, Gerhard Popp, Andreas Rausch, and Wolfgang Schwerin. The Living Software Development Process. *Software Quality Professional Journal*, 5(3), Juni 2003.

[Gog00] Martin Gogolla. Graph Transformations on the UML Metamodel. In Jose D. P. Rolim, Andrei Z. Broder, Andrea Corradini, Roberto Gorrieri, Reiko Heckel, Juraj Hromkovic, Ugo Vaccaro, and Joe B. Wells, editors, *Proc. ICALP Workshop Graph Transformations and Visual Modeling Techniques (GVMT'2000)*, pages 359–371. Carleton Scientific, Waterloo, Ontario, Canada, 2000.

[Gre01] Jack Greenfield. UML Profile For EJB. Rational Software Corporation, Public Draft, May 2001.

[Gro02] Object Managment Group. "Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/2002-04-10, April 2002.

[Hil00] David Hilbert. 'Mathematische Probleme. Votrag, gehalten auf dem internationalen Mathematiker-Kongress zu Paris.'. In *Nachrichten der Königlichen Gesellschaft der Wissenschaften zu Göttingen*, pages 253–97. 1900. Translation, incorporating subsequent emendations and additions, by Mary Winston Newson in *Bulletin of the American Mathematical Society* (Ser. 2) 8 (1902): 437–79. Partially reprinted in Ewald (1996): 1096–105.

[HNS99] Christiane Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley Object Technology Series. Addison-Wesley Longman Inc., 1999.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[ISO95] ISO/IEC. Open distributed processing reference model - part 3: Architecture. International Standard, ITU-T Recommendation, 1995.

[JBR99] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *ObjectOriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Publishing Company, Wokingham, England, 1992.

[JS04] Raphael Jolly and Elefterios Stamatogiannakis. jscl-meditor - java symbolic computing library and mathematical editor. Project Hompage: `http://jscl-meditor.sourceforge.net/`, 2004.

[KE97] A. Kemper and A. Eickler. *Datenbanksysteme - Eine Einführung*. R. Oldenburg Verlag, München, Wien, 2., aktualisierte und erweiterte auflage edition, 1997.

[kog04] Kogito Homepage. `http://www.kogito.org`, 2004.

[Kov03] Jano Kovats. Erweiterung eines graphischen UML-Werkzeugs zur Spezifikation von Modell-Transformationen. Bachelor-Arbeit an der Technischen Universität München, September 2003.

[Kru00a] P. B. Kruchten. *The Rational Unified Process. An Introduction*. Addison-Wesley, 2000.

[Krü00b] Ingolf Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[Kui03] Arnout J. Kuiper. Sun J2EE Pattern Validation OptimalJ, May 2003. Sun Reference: PS/C1407006-200305052.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Object Technology Series. Addison Wesley Professional, April 2003.

[Lef94] M. Lefering. Development of incremental integration tools using formal specifications. Technical Report AIB 94-02, RWTH Aachen, Aachen, 1994.

[LS96] M. Lefering and A. Schürr. Specification of Integration Tools. In Manfred Nagl, editor, *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*, pages 324–334, Berlin, 1996. Springer.

[Lyo98] Andrew Lyons. UML for Real-Time Overview. White Paper, Rational Software Corporation, April 1998.

[Mar04] Frank Marschall. The BOTL Homepage. Project Hompage: `http://www4.informatik.tu-muenchen.de/~marschal/botl/`, 2004.

[MB03] Frank Marschall and Peter Braun. Model Transformations for the MDA with BOTL. In Arend Rensink, editor, *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*, Enschede, The Netherlands, June 2003. Univeristy of Twente. CTIT Technical Report TR-CTIT-03-27.

[Mic03] Michael Gnatz and Frank Marschall and Gerhard Popp and Andreas Rausch and Wolfgang Schwerin. Enabling a living software development process with process patterns. Technical Report TUM-I0310, Technische Universität München, Institut für Informatik, 2003.

[Mic04] Sun Microsystems. Enterprise JavaBeans Technology. `http://java.sun.com/products/ejb/`, May 2004.

[Min01] Mark Minas. *Spezifikation und Generierung graphischer Diagrammeditoren*. Habilitationsschrift, Universität Erlangen-Nürnberg, Aachen, 2001. Shaker-Verlag.

[MM03] J. Miller and J. Mukerji. MDA Guide Version 1.0, May 2003.

[MS03a] Frank Marschall and Maurice Schoenmakers. Classifying Requirement Conflicts for Multi-Stakeholder Distributed Systems. In *Proceedings of the Workshop on Requirements Engineering in Open Systems at the 11th IEEE International Requirements Engineering Conference*, September 2003.

[MS03b] Frank Marschall and Maurice Schoenmakers. KOGITO - Knowledge Management in Requirements Engineering. In *Proceedings of the KnowTech 2003, (5. Konferenz zum Einsatz von Wissensmanagement in Wirtschaft und Verwaltung)*, October 2003.

[MS03c] Frank Marschall and Maurice Schoenmakers. Kogito document artifacts. Project Report Version 0.4, 2003.

[MS03d] Frank Marschall and Maurice Schoenmakers. Towards model-based Requirements Engineering for web-enabled B2B Applications. In *Tenth IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2003)*, pages 312–320, Los Alamitos, California, April 2003. IEEE Computer Society.

[Nov04] Novosoft. Novosoft metadata framwork and uml library. Project Hompage: `http://nsuml.sourceforge.net/`, 2004.

[NS96] M. Nagl and A. Schürr. Software Integration Problems and Coupling of Graph Grammar Specifications. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proceedings of the 5th Int. Workshop on Graph Grammars and Their Applications in Computer*

Science, number 1073 in Lecture Notes in Computer Science, pages 155–169, Berlin, 1996. Springer.

[NW01] A. Naumenko and A. Wegmann. MDA and RM-ODP: Two approaches in modern ontological engineering. Technical Report DSC/2001/047, EPFL-DI-ICA, August 2001.

[OMG00] OMG. The Common Object Request Broker: Architecture and Specification, 2000.

[OMG02a] OMG. Corba Components Version 3.0. OMG Adopted Specification formal/02-06-65, June 2002.

[OMG02b] OMG. OMG Meta Object Facility (MOF), April 2002.

[OMG02c] OMG. OMG Unified Modeling Language Specification (Action Semantics), Januar 2002. Final Adopted Specification.

[OMG02d] OMG. UML Profile for CORBA Specification, Version 1.0. OMG Specification formal/02-04-01, April 2002.

[OMG02e] OMG. UML Profile for Enterprise Distributed Object Computing Specification (EDOC). OMG Adopted Specification ptc/02-02-05, February 2002.

[OMG04] Inc. Object Management Group. Object Management Group Homepage. `http://www.omg.org`, 2004.

[ORM01] OMG Architecture Board ORMSC. Model Driven Architecture (MDA). OMG Document ormsc/2001-07-01, July 2001.

[Por80] M. Porter. *Competitive Strategy: Techniques for Analyzing Industries and Competitors*. Free Press, 1980.

[Rei82] Wolfgang Reisig. *Petrinetze - Eine Einführung*. Springer-Verlag, Berlin, 1982.

[Rit02] Beate Ritterbach. Meta-Modelle in der Objektorientierung: Teil 1. `http://www.logimod.de/MetaModelleTeil1.pdf`, Dezember 2002. Foliensatz zum Vortrag im Arbeitskreis Objektorientierung.

[Roy70] W. W. Royce. Managing the Development of Large Software System: Concepts and Techniques. In *Proceedings of the 1970 WESCON*, 1970.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.

[RS97] J. Rekers and Andy Schurr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[RVR+03] Alejandro Ramirez, Philippe Vanpeperstraete, Andreas Rueckert, Kunle Odutola, Jeremy Bennett, and Linus Tolke. *ArgoUML User Manual - A tutorial and reference description*, August 2003. Version 0.14.

[Sae02] Motoshi Saeki. Role of model transformation in method engineering. In A. Banks Pidduck et al., editor, *CAISE 2002*, number 2348 in LNCS, pages 626–642. Springer-Verlag Berlin Heidelberg, 2002.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programing*. Series in Discrete Mathematics. Wiley, 1986.

[Sch88] Andy Schürr. Modellierung und Simulation komplexer Systeme mit PROGRESS. In W. Ameling, editor, *Proc. 5. Symp. Simulationstechnik, Aachen*, number 179 in Informatik-Fachberichte, pages 84–91, Berlin, September 1988. Springer-Verlag.

[Sch91] Andy Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: formale Definitionen, Anwendungsbeispiele und Werkzeugunterstützung*. Deutscher Universitats-Verlag, Wiesbaden, 1991.

[Sch93] Uwe Schöning. *Theoretische Informatik kurz gefasst*. BI Wissenschaftsverlag, Mannheim, Leipzig, Wien, Zürich, 1993.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *Proc. WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, number 903 in LNCS, pages 151–163. Springer, 6 1994.

[Sch97] Andy Schürr. PROGRES for Beginners, 1997.

[SCJD01] Louise Scott, Lucila Carvalho, Ross Jeffery, and John D'Ambra. An Evaluation of the Spearmint Approach to Spfotware Process Modelling. In *Software Process Technology: 8th European Workshop, EWSPT 2001*, volume 2077 / 2001 of *Lecture Notes in Computer Science*, page 77 ff. Springer-Verlag Heidelberg, Januar 2001.

[SCO04] Supply-chain council homepage. `http://www.supply-chain.org/`, 2004.

[Sne98] H. M. Sneed. *Objektorientierte Softwaremigration*. Addison-Wesley, 1998.

[Sol00] Richard Soley. Model Driven Architecture. OMG White Paper Draft 3.2, Object Management Group (OMG), November 2000.

[Som95] Ian Sommmerville. *Software Engineering*. Addison-Wesley Longman Limited, Harlow, England, 5 edition, 1995.

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Phillips. Model-Based Development. Technical Report TUM-I0204, Institut für Informatik, Technische Universität München, Mai 2002.

[Spr03] Jonathan Mark Sprinkle. *Metamodel Driven Model Migration*. Dissertation, Faculty of the Graduate School of Vanderbilt University, Nashville, Tennessee, August 2003.

[SR98] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. White Paper, Rational Software Corporation, 1998.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien, 1973.

[Sun04] Sun Microsystems, Inc. J2EE Patterns Catalog. `http://java.sun.com/blueprints/patterns/catalog.html`, 2004.

[TM04] Inc. The MathWorks. MATLAB 6.5. Product Hompage: `http://www.mathworks.com/products/matlab/`, 2004.

[Tod03] Gueorgui Todorov. Realisierung einer Java-Komponente zur Inspektion Objekttransformationsregeln. Systementwicklungsprojekt an der Technischen Universität München, August 2003.

[Tra04] Rumen Traykov. Realisierung einer Java-Komponente zur Inspektion von Objekttransformationsregeln. Systementwicklungsprojekt an der Technischen Universität München, März 2004.

[TW95] R. Taylor and A. Wiles. Ring-theoretic properties of certain hecke algebras. *Annals of Mathematics*, 141(3):553–572, 1995.

[UN/01] UN/CEFACT. UN/CEFACT's Modelling Methodology. Draft, November 2001.

[VGP01] Dániel Varró, Szilvia Gyapay, and András Pataricza. Automatic transformation of UML models for system verification. In Jon Whittle et al., editors, *WTUML'01: Workshop on Transformations in UML*, pages 123–127, Genova, Italy, April 7th 2001.

[VIC04] Voluntary interindustry commerce standards association homepage. `http://www.vics.org/`, 2004.

[Wei97] Gerald M. Weinberg. *Quality Software Management: Anticipating Change*, volume 4. Dorset House, New York, Mai 1997.

[Wil95] A. Wiles. Modular Elliptic Curves and Fermat's Last Theorem. *Annals of Mathematics*, 141(3):443–551, 1995.

[Wil03a] Edward D. Willink. The UMLX Language Definition, June 2003. Working Draft.

[Wil03b] Edward D. Willink. UMLX : A graphical transformation language for MDA. In Arend Rensink, editor, *Model Driven Architecture: Foundations and Applications*, pages 13–24. University of Twente, 2003.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A FamilyBased Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., August 1999.

[WR04] Inc Wolfram Research. Mathematica 5. Product Hompage: `http://www.wolfram.com/products/mathematica/index.html`, 2004.

[XMI99] XML Metadata Interchange (XMI). OMG Document ad/99-10-02, 1999.

[XML00] Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, October 2000.

[XSL99] XSL Transformations (XSLT) Version 1.0. W3C Recommendation, 11 1999.

Index

$UGL_{validSrc}$, 112
EQATTS, 125

Aktivitätsbeschreibung, 26
Anwendbarkeit, 110
    einer Regel, 110, 130
    eines Regelwerks, 110, 130
Artefakte, 25
    Integration, 42, 201
Artefakttyp, 25
Assoziation
    Klassenassoziation, 54
    Objektassoziation, 59
    Objektvariablenassoziation, 73
attDependsOn, 117
Attribut-Gleichungssystem, 125

Beschreibungstechnik, 43
    orthogonl, 46

cannotCreateSame, 142, 145
createSameObj, 141, 145
createsAsso, 101
createsObj, 101
createsValidFragments, 113, 115

dependsOn, 116
determines, 117, 118
    reflexiv, 117
    transitiv, 117

EDOC, 180

getCard, 64
getova, 74
GL, 86
GL', 123
GMT, 270
Graphgrammatiken, 13

GReAT, 267

Instanzierbarkeits-Gleichungssystem, 65
Integrationstransformation, 46

Kardinaltiät
    Objektassoziation, 60
    Objektvariablenassoziation, 73
Klasse, 52
    instanzierbar, 67
    Konfigurationen, 64
Klassenassoziation, 54
    konsistent, 54
Klassenbelegung, 54
KOGITO, 179
    Artefakte, 188
        Integration, 201
    Business Requirements Modell, 184, 185
    Konzeptuelles Metamodell, 182
    Konzeptuelles Modell
        Transformation, 191
    Prozessmuster, 216
    Requirements Analysis Modell, 186
Konfiguration einer Klasse, 64
    instanzierbar, 67
Konsistenz
    Klasse, 53
    Klassenassoziation, 54
    Klassenbelegung, 54
    Modellfragment, 83
    Objekt, 58
    Objektassoziation, 60
    Objektbelegung, 59
    Objektvariable, 72
    Objektvariablenassoziation, 73
    Objektvariablenbelegung, 73
Konzeptuelles Metamodell, 28
Konzeptuelles Modell, 28

Transformation, 38, 191

lbCard, 164
lbVarCard, 162, 164

Match, 83
maxCard, 151, 152
maxCreateSameObj, 142
maxmatch, 146, 147
maxRedundant, 135, 139
maxRelevant, 135, 141
maxVarCard, 152, 155
MDA, 9
Merge, 95, 97
assoziativ, 97
kommutativ, 97
mergeable, 92, 93, 95
kommutativ, 93
Meta Object Facility (MOF), 6
Metametamodell, 24
Metamodell, 24
instanzierbar, 63, 67
isomorph, 57
objektorientiertes, 56
Metamodellkonformität, 40, 131, 166
mft, 89
minCard, 160, 161, 164
minmatch, 161
minVarCard, 161, 164
Model Driven Architecture, 9
Modell, 23, 58
Äquivalent, 40
Abstraktion, 40
Komposition, 45
konformes, 62
konsistente Erweiterung, 48
objektorientiertes, 61
Teilmodell, 47
Verfeinerung, 40
Modellfragment, 83
erzeugen gültiger, 111
isomorph, 167
Obtergrenzenkonformität, 132
Teilmodellfragment, 97
Untergrenzenkonformität, 132
Modellfragment-Match, 83, 133
Modellfragment-Match-Menge, 84
Modellfragmentrelation, 86
Modellfragmenttransformation, 89
Modelltransformationsregel, 75
Modelltransformationsregelwerk, 75
Modellvariable, 74
isomorphe, 136
Substrukturen, 136
MOF, 6

Notation, 28
numAssos, 133

OBerge
kommutativ, 94
Obergrenzenkonformität, 132, 133, 152, 155
Objekt, 58
Objektassoziation, 59
Objektbelegung, 59
zusammenführen, 94
Objektvariable, 71
deterministisch, 115, 119
konfliktfrei, 119, 120, 122, 127
Objektvariablenassoziation, 73
redundante, 135
Objektvariablenassoziationsende, 73
Objektvariablenattribut, 72
Objektvariablenbelegung, 73
OBmerge, 94, 95
assoziativ, 95
oppositeEnd, 56
OPTIMIX, 267
OVP, 137

potConflicting, 122
Produktmodell, 26
PROGRES, 263
Prozessmodell, 25, 26
Prozessmuster, 33
Aktivitätsmetamodell, 35
Produktmodell, 34

Quell-Matches, 168
Quellmetamodell, 76
Quellmodell, 76, 98
Quellmodellvariable, 75
QVT, 11

redundant, 138, 139

Regel, 75
    Anwendbarkeit, 130
    Anwendung, 98
    constraint-behaftet, 90
    lbConform, 159
    Umkehrregel, 78
    Untergrenzenkonformität, 156, 165
Regelanwendung, 98
Regelwerk, 75
    Anwendbarkeit, 130
    Anwendung, 99
    bijektiv, 168, 170
    Komposition, 101
    match-bijektiv, 169, 170
    Metamodellkonformität, 131, 166, 171
    Obergrenzenkonformität, 133, 155
    streng bijektiv, 168, 169
    streng match-bijektiv, 168, 169
    Umkehrregelwerk, 78
    Untergrenzenkonformität, 156, 165
Regelwerksanwendung, 99
relevant, 140, 141
Rewrite-Regelwerksanwendung, 106

Sicht, 29
    orthogonal, 46
srcMatches, 168
srcModel, 98
Substrukturen, 136

Teilmodell, 47
Term
    TermTB, 71
    Ahnliche Identifikatorterme, 91
    Attributterm, 72
    Identifikatorterm, 72
    Identifikatorterme $Term_{\mathbb{ID}}$, 71
Transformationsregel, 75
Typ, 51
Typbelegung, 52

ubVarCard, 153, 155
Umkehrregel, 78
Umkehrregelwerk, 78
UML
    BOTL Profil, 78
UML Profile for Enterprise Distributed Object Computing Specification (EDOC), 180
UMLX, 270
UMM, 181
UN/CEFACT Unified Modelling Methodology (UMM), 181
Untergrenzenkonformität, 132, 156, 165

Variablennamen, 71
varLbConform, 158, 159
Verfeinerungsabbildung, 40
Verfeinerungskomposition, 46

Zielmetamodell, 76
Zielmodell, 76
Zielmodellvariable, 75