Verteilte Anwendungssysteme als Komposition klassifizierter Softwarebausteine:

Ein Komponenten-basierter Ansatz zur Generativen Softwarekonstruktion

Frank Griffel

Vom Fachbereich Informatik
der Universität Hamburg
als Teilleistung zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

Betreuer und 1. Gutachter:	Prof. Dr. W. Lamersdorf
2. Gutachterin:	Prof. Dr. E. Horn
3. Gutachter:	Dr. D. Moldt

Tag der Disputation: 12. Juli 2001

Für Tim, der kleinen Komponente,
die unser System vervollständigt hat.

Vorwort

Mit etwas »Software-Science-Fiction«-Phantasie, könnte man sich Strukturen aus kleinen, vollständig geprüften Bausteinen vorstellen. [...] Verlassen wir also die Zukunft und wenden und wieder der rauhen, gegenwärtigen Wirklichkeit zu.

Friedrich Haugg, Software-Engineering, 1983

Ob wir uns schon in der oben angesprochenen Zukunft befinden ist fraglich, obwohl einiges dafür zu sprechen scheint: Die *komponentenbasierte Softwareentwicklung* ist in aller Munde und weckt vielversprechende Erwartungen wie eine verbesserte Wiederverwendung, Überprüfbarkeit und Wartbarkeit, eine höhere Produktivität und Qualität in der Erstellung von Software. Doch zum jetzigen Zeitpunkt fehlt immer noch eine solide Basis auf der sich solche Erwartungen wirklich gründen könnten. Schlimmer noch, es liegt nicht einmal ein Konsens darüber vor, was eine *Komponente* eigentlich ist und welche Konzepte und Technologien für sie benötigt werden. Unzweifelhaft ist allerdings, das die *Komposition*, das heißt der Baustein-artige Zusammenbau von vorgefertigten Softwareeinheiten, das zentrale Konzept einer komponentenbasierten Sichtweise auf die Applikationserstellung bildet.

Die vorliegende Arbeit leistet einen Beitrag auf dem wünschenswerten Weg zu einer industriellen Softwareproduktion mit ihren entsprechenden Merkmalen, indem konzeptionelle Grundlagen und technische Mechanismen untersucht bzw. erarbeitet werden, die eine weitgehend Computer-gestützte und automatisierte Komposition vollständiger Anwendungen aus vorgefertigten, unter Umständen über Netzwerke bereitgestellten Einzelkomponenten erlauben.

Im Gegensatz zum Komponentenbegriff gilt das Konzept des *Dienstes* im Gebiet der Verteilten Systeme seit einigen Jahren als tiefgehend erörtert und gut verstanden bzw. etwa im Rahmen des Referenzmodells für offene verteilte Verarbeitung (RM-ODP) auch wohldefiniert. Die kommunikationstechnisch notwendige Betonung der *Schnittstellen* solcher Dienste hat auch zu einer deutlichen Fokussierung auf die Betrachtung von Schnittstellenkonzepten – die auch als *innere Systemgrenzen* bezeichnet werden können – in der übrigen softwaretechnologischen Diskus-

sion geführt, wo diese Konzepte mit den Grundsätzen der Objektorientierung Entsprechungen finden. Die in einer typischen Client / Server-Sicht jedoch von einem höheren Isolationsgrad geprägten und entsprechend gut abgrenzbar vorliegenden Dienste kommen einer bausteinartigen Systemsicht allerdings zunächst auf natürliche Weise näher, insbesondere wenn deren *gemeinsame* Nutzung – im Sinne eines Mehrwertdienstes – angestrebt wird.

Die Ausgangssituation der vorliegenden Arbeit liegt dabei in der Forderung nach einer *direkten Verzahnung* der fachlichen Modellierung solcher Systeme mit der Realisierung (Implementation) im Rahmen des Entwicklungsprozesses zur Erstellung von Softwarelösungen. Der Autor ist der Ansicht, dass erst ein solch direkter Übergang die wünschenswerten Aspekte der theoretischen Behandlung von Softwaremodellen für die konkrete Entwicklungspraxis *erhält*. Die übliche Abbildung, die fach- bzw. domänenspezifische Konzepte erfahren, sobald sie von Entwicklern mehr oder weniger methodisch in konkrete Software überführt wird, ist in der Regel nicht verlustfrei. Dies führt in der Realität beispielsweise häufig dazu, dass im Modell noch formal zugesicherte Randbedingungen und Annahmen über die Software in der realen Applikation nicht mehr wirksam sind und somit etwa nicht zur Robustheit, Wartbarkeit und Portierbarkeit der erstellten Lösung beitragen können.

Daher resultiert aus dem Wunsch nach möglichst wenigen expliziten »Zwischenphasen« im Entwicklungszyklus, die Forderung nach einer weitgehenden *Automatisierung* der Softwareentwicklung. Ziel dieser Arbeit ist daher die Untersuchung der Möglichkeit eine *modellgetriebene Anwendungskonstruktion* zu etablieren und deren Randbedingungen und Implikationen zu betrachten. Im Vordergrund steht dabei der praktische Nachweis, dass die dabei zu Grunde gelegte generative und Architekturzentrierte Sicht der Anwendungserstellung sinnvoll realisiert werden und einen wesentlichen Beitrag zu einer produktiven Entwicklung von Anwendungssystemen leisten kann.

Wesentliche These ist hierbei, dass eine komponentenbasierte Sichtweise ausgezeichnet geeignet ist, um das gewünschte Konstruktionsprinzip im Rahmen der Leitbilder »Produktlinien« und »Anwendungsfamilien« realisieren zu können. Die Arbeit kann dabei selbstverständlich nicht den gesamten Themenkomplex abdecken, sondern konzentriert sich auf die beiden Schwerpunkte der *Typisierung* und der *Adaption* von Softwarekomponenten als Grundlage für die Funktionsweise automatischer Konstruktionsmechanismen.

Diese Schwerpunkte werden begleitet von der Betrachtung von Werkzeugen zur fachnahen Architekturspezifikation und der Technologie-unabhängigen Generierung der entstehenden Softwareapplikationen auf der technischen Ebene sowie konsistenzerhaltenden Kompositionsmodellen auf der theoretischen Ebene.

Danksagung

Mein Dank gilt zunächst Frau Prof. Dr. Erika Horn für die ausgesprochen kompetente und präzise Durchsicht meiner Arbeit und die entsprechend wertvollen Kommentare. Dieser Dank ist besonders herzlich, da Frau Prof. Horn das Koreferat unter besonderer persönlicher Belastung auf sich genommen hat.

Herrn Dr. Daniel Moldt danke ich dafür, dass er trotz vieler »kleiner Komponenten« die Zeit gefunden hat, meine Arbeit mit dem entscheidenden Blick für das Detail durchzusehen und zahlreiche korrigierende Anmerkungen zu liefern.

Besonderer Dank gilt Herrn Prof. Dr. Winfried Lamersdorf, der mein akademisches Leben über viele Jahre geprägt und gefördert hat. Ohne seine Weitsicht auf die Konvergenz und Synergie unterschiedlicher Themen und die resultierende kreative und befruchtende Vielfalt seiner Gruppe »Verteilte Systeme« wäre meine Arbeit nicht möglich gewesen.

Entsprechend gilt mein Dank auch allen derzeitigen und ehemaligen Kollegen, insbesondere Herrn Dr. Michael Merz für die eine oder andere Energiespende durch intensives »Schraubendrehen« und Herrn Dr. Mark Tu für das entscheidende Quantum asiatischer Gelassenheit und Weisheit – und natürlich die »Röllchen« .

Danken möchte ich insbesondere auch den mir anvertrauten Diplomanden Nail Araz, Michael Awizen, Thomas Benthien, Henning Brandt, Nicola Broschei, Alexei Chmelev, Joachim Ehm, Sophia Fleischer, Marius Lapok, Nguyen-Thinh Le, Christian Martens, Tobias Neubert, Thomas Plümpe, Stefan Reich, Oliver Schmelzle, Martin Schmidt und Dapeng Wang, deren engagierte Mitarbeit und Diskussionsfreude erheblich zu den hier vorgestellten Ergebnissen meiner Arbeit beigetragen haben, sowie insbesondere Andreas Häming, Malte Moehl, Thomas Kröhling, Dirk Tetau und Christian Zirpins, deren Diskussionen und Diplomarbeiten in die vorliegende Darstellung eingeflossen sind und ohne deren tatkräftige und geduldige Unterstützung bei den umfangreichen Implementationsarbeiten eine derart komplexe Entwicklungsumgebung nicht in der zur Verfügung stehenden Zeit hätte realisiert werden können. Besonderer Dank gilt auch

Andreas Bernhardt und Dejan Djokić für deren Konsolidierung und Integration meiner frühen Prototypen zu einer praxistauglichen Gesamtumgebung, die es auch ermöglicht hat, industriellen Anwendern das Prinzip einer Generativen Softwarekonstruktion komplexer, verteilter Anwendungssysteme überzeugend nahezubringen.

Frank Griffel *Hamburg, im Juli 2001*

Inhaltsverzeichnis

1 Einführung 1
1.1 Motivation 1
1.1.1 Historie: Die »Softwarekrise« 2
1.1.2 Verteilung und Interaktion 4
1.1.3 Konnektoren zur Adaption 6
1.1.4 Verhältnis zur Objektorientierung 8
1.1.5 Vorgehensmodelle und -prozesse 13
1.1.6 Charakterisierung des Komponentenbegriffs . . . 20
1.2 Zielsetzung und Gang der Untersuchung 25

2 Modell 31
2.1 Technisch motivierte Aspekte 31
2.1.1 Komponenten als Basisbausteine einer Softwareapplikation 32
2.1.1.1 Struktur, Art und Verhalten 32
2.1.1.2 Funktionalität und Granularität 37
2.1.2 Integration und Interoperabilität 38
2.1.2.1 Komposition 39
2.1.2.2 Interoperabiltität 41
2.1.2.3 Nebenläufigkeit 43
2.1.2.4 Verteilung 43
2.1.2.5 Übergeordnete Strukturen und Rahmenwerke 44
2.1.3 Formalisierung 45
2.2 Entwicklungsmethodik 46
2.2.1 Komponentenentwicklung 47
2.2.2 Anwendungsentwicklung 47
2.3 Struktur- und Modellsicht des Komponentenbegriffs . . . 49
2.3.1 Vorüberlegungen 49
2.3.1.1 Ausprägungen eines Komponentenmodells 49
2.3.1.2 Anforderungen 50
2.3.1.3 Methodik 54
2.3.1.4 Integration einer Refinement-Sicht 55

2.3.2 Die Modellebene 56
2.3.2.1 Strukturmodelle 56
2.3.2.2 Verhaltensmodelle 65
2.3.2.3 Metamodelle 67
2.3.3 Die Beschreibungsebene 68
2.3.3.1 Einflußfaktoren 68
2.3.3.2 Charakter einer Spezifikation 71
2.3.3.3 Formalisierung 80
2.3.4 Entwurf eines Komponentenmodells 88
2.3.4.1 Realisierung der Modellebene 89
2.3.5 Verhaltensbeschreibung mittels Temporallogik . 97
2.3.5.1 PLTL basierte Formalisierungskonzepte des Komponentenmodells 100
2.3.6 Flexible Beschreibungsformen mittels Feature-Logik 110
2.3.6.1 Feature-Logik basierte Formalisierungskonzepte eines Komponentenmodells 110
2.3.7 Übergreifende Aspekte formaler Konzepte 115
2.3.7.1 Kombinationsmöglichkeiten deskriptiver und temporaler Logiken 115
2.3.7.2 Typisierungsansätze 118
2.3.7.3 Überblick und Bewertung der Ergebnisse 121

3 Adaption 123
3.1 Konzepte statischer und dynamischer Komponentenadaption 123
3.1.1 Komponentenkompatibilität 124
3.1.2 Komponentenanpassung 126
3.1.2.1 Statische Anpassung 126
3.1.2.2 Dynamische Anpassung 126
3.1.2.3 Anpassungsmethoden 127
3.1.2.4 Bewertung hinsichtlich des Einsatzzeitpunkts 128
3.1.2.5 Bewertung hinsichtlich Einsatzgebiet . . 128
3.1.2.6 Abschließende Bewertung 130
3.2 Ein Modell dynamischer Komponentenadaption 131
3.3 Begriffseinführung und Voraussetzungen 131
3.3.1 Kollaborationsspezifikation 132
3.3.2 Technische Anforderungen an die Systemumgebung 134
3.4 Kompatibilitätsprüfung 135
3.4.1 Konzept 135

3.4.2 Kompatibilität der Schnittstellenspezifikationen . 136
3.4.3 Kompatibilität der Protokolle 136
3.4.4 Mehrdeutigkeit der Kompatibilitätsprüfung 139
3.4.5 Unspezifizierter Empfang 139
3.4.6 Verklemmung (deadlock) 140
3.4.7 Endlosschleife 141
3.5 Laufzeitsynchronisation 142
3.6 Algorithmus zur Kompatibilitätsprüfung 144
3.6.1 Rückgabeinformationen 144
3.6.2 Ablauf des Prüfungsalgorithmus 145
3.7 Adapter 147
3.7.1 Motivation 147
3.7.2 Konzeption des Adapters 150
3.7.3 Formale Beschreibung 151
3.7.4 Anwendung des formalen Konzepts 152
3.7.5 Praktische Umsetzung 153
3.7.6 Laufzeitverhalten 155
3.7.7 Kritische Anmerkungen 156
3.8 Prüfungsalgorithmus der Adapterkorrektheit 157
3.8.1 Rückgabeinformationen 157
3.8.2 Unterscheidung schwacher und starker Kompatibilität 158
3.8.3 Ablauf des Prüfungsalgorithmus 158
3.9 Semiautomatische Generierung von Adaptern 160
3.9.1 Eigenschaften von Adaptern 160
3.9.2 Syntax der Meta-Adapterspezifikation 161
3.9.3 Algorithmus zur automatischen Generierung eines Adapterprotokolls 163
3.9.3.1 Voraussetzungen 165
3.9.3.2 Ablauf des Algorithmus 166
3.9.4 Ergänzung 167
3.10 Attributbasierte Adaption 169
3.11 Bewertung 172

4 Klassifikation 177
4.1 Vom Modell zur Laufzeitkomponente 177
4.1.1 Ansätze der Typisierung 178
4.1.1.1 Verhältnis zum Spezifikationsbegriff . . . 180
4.1.1.2 Gegenstände der Typisierung 182
4.2 Typmanagement 182
4.3 Konzeption des Typmodells 184
4.3.1 Semantischer Typ 185
4.3.2 Syntaktischer Typ 187
4.3.3 Formalisierung des Typmodells 188

4.3.4 Protokollrepräsentation durch endliche Automaten 192
4.3.4.1 Eindeutigkeit von Protokollen 193
4.4 Beziehungen zwischen Typen 194
4.4.1 Klassifikation und Typisierung 194
4.4.1.1 Typ vs. Klasse 195
4.4.1.2 Beziehungen im Typmodell 197
4.4.2 Typarten 198
4.4.2.1 Affinität der Typarten zum Basistyp . . . 199
4.4.3 Assoziationen im Typmodell 201
4.4.3.1 Semantische Ebene 202
4.4.3.2 Nachrichtenebene 203
4.4.3.3 Syntaktische Ebene 204
4.4.3.4 Benutzerdefinierte Assoziationen 205
4.5 Assoziationsauswertung im Typmodell 206
4.5.1 Semantische Ebene 206
4.5.1.1 Definition Typgleichheit 206
4.5.1.2 Definition Typart-1 (Subtyp) 207
4.5.1.3 Definition Typart-2 (Verhaltenstyp) 207
4.5.1.4 Definition Typart-2 von Typart-1 207
4.5.1.5 Definition Kompatibilität 208
4.5.2 Nachrichtenebene 208
4.5.2.1 Definition Typgleichheit 209
4.5.2.2 Definition Typart-1 (Subtyp) 209
4.5.2.3 Definition Typart-2 (Verhaltenstyp) 210
4.5.2.4 Definition Typart-2 von Typart-1 210
4.5.2.5 Definition Kompatibilität 211
4.5.3 Syntaktische Ebene 211
4.5.3.1 Definition Typgleichheit 212
4.5.3.2 Definition Typart-1 (Subtyp) 212
4.5.3.3 Definition Typart-2 (Verhaltenstyp) 213
4.5.3.4 Definition Typart-2 von Typart-1 213
4.5.3.5 Definition Typart-3 (technischer Typ) . . 214
4.5.3.6 Definition Kompatibilität 214
4.6 Konformität vs. Kompatibilität 214
4.6.1 Substitutionskandidaten 215
4.6.2 Interaktionspartner 216
4.7 Adapter 217
4.7.1 Vollständige Kompatibilität 218
4.8 Mono- vs. Polymorphismus: Komponentenrollen 220
4.8.1 Konsequenzen für ein Typmanagement 221
4.8.2 Adapter 222
4.9 Kompositionstyp 223
4.10 Typdomänen 223

4.11 Bewertung 224

5 Konstruktion 227
5.1 Motivation und Grundkonzeption 227
5.1.1 Überblick 231
5.2 Produkte und Architekturen eines Anwendungsbereichs 232
5.2.1 Produktlinien 232
5.2.2 Produktfamilien 234
5.2.3 Architekturbeschreibungssprachen 235
5.2.3.1 Bewertung und Abgrenzung 237
5.3 Aufbereitung eines Anwendungsbereichs 238
5.3.1 Domänenanalyse 239
5.3.1.1 Umfang der Domäne 240
5.3.1.2 Kategorisierung des Umfanges und der Domänen 240
5.3.1.3 Beziehungen zwischen Domänen 241
5.3.1.4 Fachwissen 241
5.3.1.5 Domänenmodell 242
5.3.1.6 Merkmale 242
5.3.2 Domänendesign 244
5.3.2.1 Architektur 245
5.3.2.2 Produktionsplan 245
5.3.3 Domänenimplementation 246
5.3.4 Feature Oriented Domain Analysis (FODA) 246
5.3.5 Fazit 248
5.4 Software-Generatoren 249
5.4.1 Motivation für Generatoren 249
5.4.2 Produktkonfiguration (Customizing) 251
5.5 Generative Programmierung und verwandte Konzepte . 252
5.5.1 Aspektorientierte Programmierung (AOP) 253
5.5.1.1 Der Begriff des Aspekts 254
5.5.1.2 Motivation der Aspektorientierung 254
5.5.2 Adaptive Programmierung (AP) 255
5.5.3 Bewertung von AOP und AP 257
5.5.4 Aspektorientierung vs. generative Programmierung 259
5.5.5 Aspekte vs. Komponenten 260
5.5.6 Transformationssysteme 260
5.5.7 Transformationssysteme vs. Generatoren 261
5.5.8 Intentional Programming 262
5.6 Prinzip eines Software-Generators 262
5.6.1 Generatoren als Compiler 263
5.6.2 Frameworks und Generatoren 264
5.6.3 Entwicklungsprozesse und Generatoren 265

5.7 Beispiele generativer Ansätze 265
5.7.1 GenVoca 265
5.7.1.1 Subjektive Schnittstellen 268
5.7.1.2 Validierung von Kompositionen 269
5.7.2 Jakarta 270
5.7.3 C++ Templates 270
5.7.3.1 Das Template-Konzept 271
5.7.3.2 Vorgehensweise 271
5.7.3.3 Beispielanwendung von C++-Templates 272
5.7.3.4 Bewertung 273
5.7.4 Convergent Architecture 273
5.8 Konzeption eines Deskriptor-gesteuerten Generators . 274
5.8.1 Einordnung 274
5.8.2 Motivation 276
5.8.3 Grundidee 278
5.8.4 Phasen der Generierung 279
5.8.5 Verteilung der Informationen 281
5.9 Komponentendeskriptoren: XML als Ausdrucksmittel . . 282
5.9.1 Deskriptoren zur Schnittstellenbeschreibung . . . 283
5.9.2 DTD zur Architekturrepräsentation 285
5.9.3 Klassifikation der Komponenten 287
5.9.4 Zusammenfassung 288
5.10 Auswahl der Komponenten 288
5.10.1 Auswahl der Merkmale einer Anwendung 288
5.10.2 Kompatibilität von Komponenten 289
5.10.3 Verknüpfung von Komponenten 292
5.10.4 Priorisierung der Auswahlentscheidungen für Merkmale 293
5.11 Leitbild für die Implementation 294
5.11.1 Der Generator aus Anwendersicht 294
5.12 Eine Beispieldomäne 298
5.12.1 Unterscheidungsmerkmale von Taschenrechnern 299
5.12.2 Die Komponententypen eines Taschenrechners 300
5.12.3 Interaktionsmuster der Taschenrechner-Domäne 301
5.12.4 Anwendungsstruktur 303
5.12.5 Abhängigkeiten und Prioritäten 305
5.12.6 Behandlung optionaler Komponenten 305
5.13 Zusammenfassung 307
5.13.1 Generative Softwarekonstruktion 307
5.13.2 Generierungsphasen 308
5.13.3 Einordnung und Bewertung 309

5.13.4 Konzeption einer vollständigen Entwicklungsumgebung 312

6 Implementation 315
6.1 Die Adapter 315
6.1.1 Anforderungen an Komponentenentwickler 316
6.1.1.1 Kollaborationsspezifikation 316
6.1.1.2 Entfernter Methodenaufruf mittels `SignaturEvent` 318
6.1.1.3 Methodenaufruf auf eine Komponente . 319
6.1.2 Systemarchitektur: Kommunikationsschema . . . 319
6.1.2.1 `Extension`-Objekte 319
6.1.2.2 Kopplungsvorgang 322
6.1.2.3 Synchronisation zur Vermeidung von Laufzeitproblemen 322
6.1.3 Kompatibilitätsprüfung 323
6.1.3.1 Initialisierung 324
6.1.3.2 Zustandspaare 325
6.1.4 Ablauf 325
6.1.4.1 Rückgabe des Resultats 327
6.1.4.2 Einbinden der Kompatibilitätsprüfung . 328
6.1.5 Adapterkomponenten 328
6.1.5.1 Einbindung in Kommunikationsablauf . 328
6.1.5.2 Deklarativer Teil 329
6.1.5.3 Implementation der Funktionalität des Adapterprotokolls 330
6.1.6 Kompatibilitätsprüfung für Adapter 333
6.1.6.1 Zustandstripel 333
6.1.6.2 Ablauf 334
6.1.6.3 Rückgabe des Resultats 335
6.1.7 Semiautomatische Generierung von Adapterprotokollen 335
6.1.7.1 Die Klasse `Meta_Adapter` 336
6.1.7.2 Die Klasse `Generate_Adapter` 337
6.1.7.3 Tuple_of_States 338
6.2 Der Typmanager 343
6.2.1 Typregistrierung 345
6.2.1.1 Typbeschreibungen in XML 345
6.2.1.2 XML-Deskriptoren 346
6.2.1.3 Erweiterung der CCM-Deskriptoren . . . 351
6.2.1.4 Parsing-Phase 356
6.2.1.5 Abbildung der XML-Deskriptoren 357
6.2.1.6 Abbildungsalgorithmus für Merkmalsstrukturen 360

6.2.2 Typoperationen 363
6.2.2.1 Typvergleich 363
6.2.2.2 Protokollvergleich 366
6.2.3 Kompatibilitätsprüfung 368
6.2.4 Typsuche 372
6.2.4.1 Basistypen und Typarten 372
6.2.4.2 Kompatible Typen 374
6.2.5 Adapter 374
6.2.5.1 Prüfung beidseitig-vollständiger Kompatibilität 375
6.2.6 Zusatzfunktionen 377
6.2.6.1 Typdomänen 378
6.2.6.2 Benutzerdefinierte Assoziationen 379
6.2.6.3 Ansicht der Typbeschreibung 379
6.2.7 API des Typmanagers 381
6.2.8 Modularisierung der Implementation 382
6.3 Der Generator 384
6.3.1 Technische Kurzbeschreibung 385
6.3.2 Auswahl der Komponenten 389
6.3.2.1 Reihenfolge der Merkmalsspezifikation 389
6.3.2.2 Auswahl einer einzelnen Komponente . 390
6.3.2.3 FeatureGUI 393
6.3.3 ComponentPool 393
6.3.4 Zusammenspiel der Komponenten bei der Merkmalsauswahl 395
6.3.5 Aufbau der Anwendungsstruktur 397
6.3.5.1 Der XML-Parser 398
6.3.5.2 Funktionsweise des XMLDocBuilder . . 398
6.3.5.3 Einschränkungen des XMLDocBuilder 402
6.3.6 Quellcode-Generierung für das Anwendungsgerüst 405
6.3.6.1 Individuelle Informationen über eingesetzte Komponenten 405
6.3.6.2 Funktionsweise des AppBuilder 407
6.3.6.3 Verknüpfung von Komponenten 409
6.3.6.4 Post-Konfiguration 410
6.3.6.5 GUISlots 410
6.3.6.6 Kompilierung und Archivierung 411
6.3.7 Erweiterung der Komponentendeskriptoren 412
6.4 Unterstützungswerkzeuge 414
6.4.1 Der Deskriptor-Editor 414
6.4.2 Der Protokoll-Editor 416
6.4.3 Der Merkmals-Editor 419

7 Resümee und Ausblick 427

A DTDs der XML-Deskriptoren 437
A.1 corbacomponent.dtd 437
A.2 syntBEType.dtd 438
A.3 syntMessageType.dtd 439
A.4 FeatureStructure.dtd 439
A.5 NFA.dtd 439

B Formale Grundlagen der Protokollkompatibilität . . . 441

C Beispiel-Protokolle des Bankszenarios 445
C.1 Kundenprotokoll 445
C.2 Bankprotokoll 446
C.3 Sparkassenprotokoll 447
C.4 Metaadapterprotokoll zur Zusammenarbeit Kunde / Sparkasse 448

D Definition exemplarischer Komponententypen 451
D.1 Kunde-Bank-Beispiel 451
D.1.1 Typbeschreibungen der Komponenten 452
D.1.1.1 Substitutionskandidaten 454
D.1.1.2 Interaktionspartner 456
D.1.2 Repräsentation der Semantik der Business Events 457
D.1.2.1 TransferRequest 458
D.1.2.2 Register 458
D.1.2.3 Unregister 459
D.1.3 Beziehungen zwischen Typen 460
D.1.3.1 Typvergleich 460
D.1.3.2 Kompatibilitätsprüfung 461
D.1.4 Adapter 463
D.1.5 Polymorphismus 465
D.2 Druckdienst-Beispiel 466
D.2.1 Typbeschreibungen der Dienste 466
D.2.1.1 Substitutionskandidaten 469
D.2.1.2 Interaktionspartner 471
D.2.2 Repräsentation der Semantik der Business Events 472
D.2.3 Beziehungen zwischen Typen 473
D.2.3.1 Typvergleich 473
D.2.3.2 Kompatibilitätsprüfung 476
D.2.4 Adapter 476
D.2.5 Polymorphismus 478

E Klassenstruktur des Typmodells 479

F Omega-Kanten 481

G Assoziationen im Typmodell 483
G.1 Semantische Ebene 483
G.2 Nachrichtenebene 484
G.3 Syntaktische Ebene 484

H Propositionale Lineare Temporale Logik 485
H.1 Lineare Zeitlinien 486
H.2 PLTL-Syntax 487
H.3 PLTL-Semantik 487
H.4 Grundbegriffe des logischen Systems 488
H.5 Verifikation 488

I Feature Logik 491
I.1 Features und Feature-Terme 491
I.2 Rollen 492
I.3 Beziehungen 493
I.4 Feature-Strukturen 494

Abbildungsverzeichnis 497

Tabellenverzeichnis 501

Literaturverzeichnis 503

1 Einführung

1.1 Motivation

Dieser Abschnitt soll einen kurzen Abriss des allgemeinen Hintergrunds der aktuellen Diskussion des Komponentenparadigmas liefern, um die Vielfalt der für den Ansatz dieser Arbeit in der weiteren Diskussion zu beachtenden Einflussfaktoren und Randbedingungen widerzuspiegeln bzw. entsprechende Trends und Wegbereiter aufzuzeigen, die den vorliegenden Ansatz inspiriert haben. Eine ausführlichere Darstellung des allgemeinen Komponentenumfeldes findet sich in [Griffel, 1998].

Es ist zunächst festzuhalten, dass sich die Forschung im Umfeld des Komponentengedankens noch nicht in einem gefestigten Stadium befindet. Allerdings existieren eine Reihe vielversprechender Ansätze und Projekte, die sich meist auf Teilbereiche des komplexen Themengebiets konzentrieren und auf die Realisierbarkeit der jeweils angestrebten Konzepte hoffen lassen. In diesem Zusammenhang stellt sich die Frage der Identifizierung und Abgrenzung relevanter Problembereiche. [Nierstrasz & Dami, 1995] heben in Bezug auf eine komponentenbasierte Entwicklung die folgenden drei Forschungsschwerpunkte hervor:

- Die Verschmelzung der gegenwärtigen Abstraktionsbegriffe von Prozesskalkülen sowie funktionalen- und objektorientierten Sprachen zu einer Konkretisierung des *Komponentenbegriffs*, der eine persistente, »first-class« Einheit mit Konzepten der Parametrisierung, Instanziierung und der Möglichkeit zur Skalierung darstellen sollte.

- Die Entwicklung von unterstützenden *Softwarewerkzeugen*, die die Manipulation partieller Konfigurationen erlauben und einen iterativen Zusammenbau von Komponenten in einer Vielzahl verschiedener Darstellungsschichten ermöglichen sollten.

- Das Finden ausdrucksstarker *Typsysteme*, die die Korrektheit und Sinnhaftigkeit von Software-Konfigurationen in für den Programmierer transparenter Weise entscheiden können.

Die vorliegende Arbeit versteht sich als Beitrag zu einer derartigen Forschungsagenda und ist unter anderem durch diese motiviert worden.

1.1.1 Historie: Die »Softwarekrise«

Obwohl *Componentware* in aller Munde ist, herrscht über deren konkrete Inhalte noch weitgehende Uneinigkeit und sowohl in Kreisen der freien Wirtschaft als auch der klassischen Wissenschaft finden heftige, zum Teil kontroverse Diskussionen darüber statt, was die Grundlagen dieser Herangehensweise ausmacht und in welchem Verhältnis sie zu anderen Sichten steht [Broy et al., 1998]. Überwiegendes Einvernehmen herrscht lediglich bei den durch eine Komponentenorientierung angestrebten Zielen, die in ihrer Essenz dem Wunsch nach einer Lösung dringender, traditioneller Probleme entstammen.

An erster Stelle steht dabei im Allgemeinen die seit langem mit bisher eher geringem Erfolg angestrebte Aktivierung des Potenzials einer *Wiederverwendung* einmal entwickelter Softwarelösungen. Damit eng einher gehen Probleme der *Anpassbarkeit*, *Flexibilität* und *Wartbarkeit* von Programmsystemen sowie deren *Evolution* (auch zur Laufzeit) über den gesamten Lebenszyklus.

Die Erstellung von Softwaresystemen ist zu einem der wichtigsten wirtschaftlichen wie gesellschaftlichen Faktoren geworden, von dem das Wohl und die Zukunft komplexer sozioökonomischer Gefüge abhängen. In der Wirtschaft ist leistungsfähige Software im Wettbewerb überlebenswichtig. Im Hinblick auf gesellschaftliche Belange wären die staatliche Verwaltung, öffentliche Dienste und innere Sicherheit genauso wie das Gesundheitswesen und der Sozialbereich handlungsunfähig [Friedrich et al., 1995].

Seit Jahrzehnten wird nun von einer *Softwarekrise*[1] gesprochen, die unter Berücksichtigung der gleichermaßen weitreichenden wie folgenschweren Wechselwirkungen innerhalb obiger Bereiche nicht als bloße Phrase abgetan werden kann. Die Proble-

[1] Heutige Entwickler verwenden ca. 70 Prozent ihrer Arbeitszeit für die Wartung und Pflege bereits bestehender Software. Die Softwarekrise ist daher eigentlich eher eine »Wartungskrise« [Gluch & Weinstock, 1998].

matik der Krise bezieht sich dabei auf die mangelnde Fähigkeit, Programme in einer dem steigenden Bedarf gerecht werdenden qualitativen und quantitativen Weise zu erzeugen. Ein wesentlicher Grund dieser Schwäche ist in der Methodik zu sehen, mit der heute Software als immer wieder grundauf neue Einzelanfertigung erstellt wird. Als vielversprechender Lösungsansatz bietet sich hier nun die Componentware-Idee von einer systematischen Wiederverwendung bewährter, qualitativ hochwertiger Teilbausteine im Sinne des Vorgehens klassischer Ingenieursdisziplinen an. Softwareerstellung soll durch diese Vorgehensweise beschleunigt werden und gleichzeitig zu verläßlicheren Ergebnissen führen. Neben dem Ziel einer adäquaten Bedarfsbefriedigung verspricht diese Strategie dann auch noch eine erhebliche Steigerung der Wirtschaftlichkeit, entspricht sie doch auf betriebswirtschaftlicher Ebene dem Wechsel von klassischer Einzelanfertigung bei Werkstattfertigung zu diversifizierter Großserienproduktion bei Fließfertigung.

Werden die Zielsetzungen der Komponentensicht und der daraus resultierende umfangreiche Forderungskatalog an diesbezügliche Techniken in ganzer Breite vor Augen geführt, werden unweigerlich Zweifel an der Realisierbarkeit geweckt, sind doch praktisch alle wesentlichen, seit langem offenen Probleme des Metiers enthalten. Es stellt sich also im Folgenden die Frage nach der entsprechenden inhaltlichen Basis eines *Komponentenparadigmas*, die in der Lage ist, den hochgesteckten Erwartungen zu genügen.

Die Hinterfragung entsprechender konzeptioneller Grundlagen komponentenbasierter Softwareentwicklung besitzt weit zurückreichende, traditionelle Wurzeln. Schon 1968 prognostizierte [McIlroy, 1968] den Wandel der Softwareerstellung von den bislang stets grundauf neuen, individuellen Einzelentwicklungen hin zu einer (Re-)Kombination vorgefertigter, bewährter Teilprogramme nach dem Vorbild traditioneller Ingenieursdisziplinen.

Seit diesen ersten Ansätzen – McIlroy dachte seinerzeit an Bausteine in Form textueller Sourcecode-Fragemente – hat sich die Idee der Komponenten kontinuierlich gewandelt. Der Begriff »Komponente« als sourcecodebasierter Programmbaustein ist stark aufgeweicht worden und einer Flut unterschiedlichster Auffassungen gewichen. Der Begriff unterliegt in seiner Auslegung heute einer großen Bandbreite möglicher Inhalte. Was eine Komponente *ist, sein kann* oder *sein sollte* reicht potenziell von sehr allgemeinen Auffassungen wie »*jegliches Artefakt der Softwareentwicklung das wiederverwendbar ist*« [Hooper & Chester, 1991] bis zu engen Konzept-, Sprach- oder

gar produktgebundenen Auslegungen wie »*Java Klasse, die den Bean-Konventionen genügt*« [Sun, 1997a]. Diese beiden Beispiele geben in etwa die Extreme der bestehenden Palette möglicher Komponentendefinitionen wieder. Eine detaillierte Diskussion der wichtigsten Ansichten und Vorschläge erfolgt weiter unten in Abschnitt 1.1.6.

Innerhalb der vorliegenden Arbeit soll der Komponentenbegriff nicht per se auf jegliche Artefakte einer Softwareentwicklung ausgeweitet werden, sondern seines Wesens nach ausführbaren Code umfassen. Die Entwicklung eines Komponentenverständnisses im Sinne von ausführbarem Code kann in enger Analogie zur historischen Zunahme des Abstraktionsgrades von Programmiersprachen gesehen werden [Griffel, 1998].

1.1.2 Verteilung und Interaktion

Das also bereits Jahrzehnte Zurückliegen des Anstosses einer Komponentendiskussion lässt die Frage aufkommen, warum gerade derzeit das Interesse an expliziten Komponentenmodellen so sehr in den Vordergrund gerückt ist. Ein erster wesentlicher Punkt ist sicherlich die Suche nach Alternativen bzw. Ergänzungen des Paradigmas der *Objektorientierung*, das in der Entwicklungspraxis bislang die Erwartungshaltung insbesondere in Bezug auf Wiederverwendung und Wartbarkeit nicht befriedigen konnte.[2]

Daneben trägt auch die Desktop-Metapher grafischer Benutzeroberflächen mit ihrem bausteinartigen Aufbau zu einem Fokus auf den Komponentenbegriff bei. Ausschlaggebende Motivation für die vorliegende Arbeit sind schließlich jedoch insbesondere die im Rahmen zunehmender Vernetzung bereitstehenden Dienste und die damit verbundenen Integrationsbemühungen heterogener verteilter Systemstrukturen. Diesem Gebiet der Client / Server- bzw. Middleware-Diskussion entstammt denn auch der Begriff der »Componentware« [Stal, 1995].

Dieser Verteilungstrend schafft sowohl die Voraussetzung als auch den Bedarf für eine neue Klasse von Softwaresystemen, die die Funktionalität verschiedener, autonomer Teile nutzen und zu neuen, komplexen Anwendungen zu kombinieren in der Lage

[2] [Smaragdakis & Batory, 1999] gehen etwa soweit, Objekte als in der Regel nicht wiederverwendbar zu bezeichnen, da sie für sich isoliert keine Bedeutung tragen. Hierin ist auch ein wichtiges Unterscheidungskriterium zur Komponentensicht zu sehen: eine Komponente trägt *kontextunabhängig* Bedeutung.

sind. In diesem Zusammenhang von wird daher auch von *Kooperationsanwendungen* [Müller–Jones et al., 1995] gesprochen.

Im Rahmen der Applikationsserver-Diskussion [Griffel, 1999a] stehen derzeit jedoch *Container*-basierte Lösungen im Vordergrund. Diese entsprechen eher einer klassischen Client-Server-Sicht, da sie Funktionalitäten (Komponenten) serverseitig zentralisieren. Komponenten dienen dabei als *innere* Strukturierung eines Containers, der als »Applikationshülle« standardisierte Schnittstellen bereitstellt. Dies erhöht die generelle Kopplungsfähigkeit bei gleichzeitiger Entkopplung der einzelnen Komponenten. Zudem stellt der Container immer wieder benötigte Systemfunktionen wie etwa Transaktionen, Verschlüsselung und Persistenz bereit. Zur konkreten Nutzung dieser zunächst generisch bereitstehehden Unterstützungsdienste erfolgt eine deklarative Festlegung der *Einsatzkonfiguration* der bereits fertiggestellten Komponenten. Die dabei benötigten Informationen haben aktuell die Diskussion um Meta-Informationen und entsprechende Meta-Modelle in der Softwareentwicklung befruchtet, deren erste pragmatische Ergebnisse heute in Form sogenannter *Komponentendeskriptoren* vorliegen, wie sie auch dem Ansatz der vorliegenden Arbeit zu Grunde gelegt werden sollen.

Losgelöst von konkreten Markt- und Technologietrends läuft der inhärente Verteilungsaspekt des Komponentenparadigmas konform zu einer interessanten visionären Ideologie der autonomen nichtdeterministischen *Interaktion*, die mit [Wegner, 1997] begründet wurde. Wegner betrachtet in kritischer Weise die Church'sche These und den resultierenden klassischen Berechenbarkeitsbegriff der Informatik, basierend auf dem Prinzip von Turingmaschinen. Insbesondere sagt er das Ende des streng deterministischen Algorithmus voraus und stellt eine Sicht, basierend auf der Interaktion freier, unabhängig agierender Instanzen, die eine maximale Form autonomer Delegation darstellen, dagegen. Unabhängig davon, ob dieser Ansatz die Diskussion der theoretischen Informatik zukünftig prägen wird oder nicht, befruchtet die Konzentration auf den Interaktionsbegriff eine konkrete Komponentensicht, da sie dem strapazierten Schnittstellenbegriff mit seinem strukturellem Charakter eine mehr verhaltensbezogene und weniger technologisch geprägte Sicht bei Seite stellt, die auch den hier vorgestellten Ansatz zur Klassifikation von Softwarebausteinen beeinflusst hat.

Komponenten interagieren mittels eines Nachrichtenaustauschs (programmtechnisch in der Regel gleichbedeutend mit einem Methodenaufruf). Während eines solchen Kommunikations-

vorganges zweier Komponenten lassen sich in eindeutiger Weise *Sender* und *Empfänger* der ausgetauschten Nachricht identifizieren (vgl. Abb. 1.1). Innerhalb der gekapselten Komponentenimplementierung können dabei sowohl beim Sender als auch beim Empfänger interne, nach außen nicht sichtbare Zustandsänderungen eintreten, die gegebenenfalls zur Weiterführung der Kommunikation durch das Senden einer weiteren Nachricht bzw. das Senden einer Antwort und somit zu einem *beobachtbaren Verhalten* führen, das im Rahmen der vorliegenden Arbeit auch als *Interaktionsmuster* bezeichnet wird.

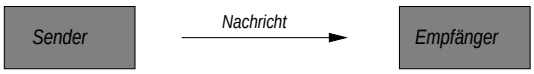


Abbildung 1.1 *Rollenverteilung während eines Kommunikationsvorganges in der Komponenteninteraktion*

1.1.3 Konnektoren zur Adaption

Eine Komponenteninteraktion im obigen Sinne setzt eine vorherige erfolgreiche Komponenten*kopplung* voraus. Komponenten »koppeln« aneinander durch das Herstellen einer Verbindung über einen oder mehrere Kommunikationskanäle zwischen ihren Schnittstellen,[3] die für die Systemumgebung den einzigen Zugangspunkt zur Funktionalität einer Komponente darstellen.

Die üblicherweise durch technische Heterogenität entstehenden Probleme verlagern sich in einer solchen *Black-Box*-Komponentensicht von der Implementierung hin zur Charakterisierung der Schnittstelle und des beobachtbaren Verhaltens. Die Schnittstellen müssen »kompatibel« sein, damit eine erfolgreiche und sinnvolle Interaktion möglich wird. Dies ist kein Problem, solange Entwickler genau wissen, wie die Schnittstellen der Komponenten beschaffen sind, mit denen die eigenen Komponenten kommunizieren sollen. In offenen und veränderlichen Systemen und vor allem auch als Folge einer stärkeren Autonomie der Komponentensicht (vgl. a. das Agentenkonzept [Griffel, 1998]) kann dies nicht ohne weiteres gewährleistet werden.

Der geschlossene Black-Box-Charakter einer Komponente soll bei einer notwendig werdenden *Anpassung* (Adaption) beibehalten werden, sodass eine *Auslagerung* entsprechend anpas-

[3] Dabei wird eine sogenannte »lose« Kopplung angestrebt, d.h. es erfolgt keine Verankerung des konkreten Kommunikationspartners im Code eines anderen. D.h. also, dass der Kopplungspartner in der Entwicklungsphase zwar *typmäßig* bekannt ist, jedoch noch kein Verweis auf eine konkrete Instanz vorgegeben wird.

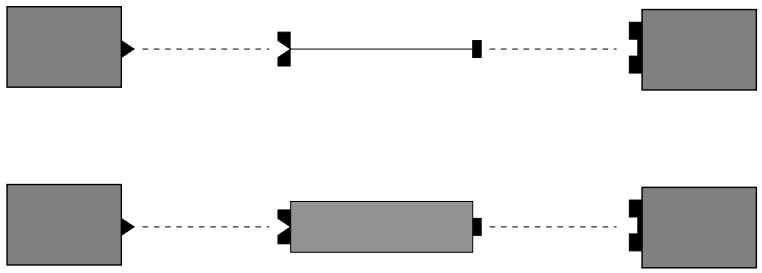


Abbildung 1.2
Konnektoren ohne oder mit implementierter Adaptionslogik ermöglichen die Überwindung von Inkompatibilitäten

sender bzw. koppelnder Mechanismen in Form von expliziten Konnektoren bzw. Adaptern sinnvoll erscheint (vgl. Abb. 1.2). Einem solchen Konnektor obliegt dann im Idealfal neben der Herstellung einer technischen Interoperablität die Einhaltung eines gemeinsamen Verständnisses bezüglich logischer Abläufe und semantischer Inhalte des wechselseitigen Informationsaustausches im Zuge der Komponenteninteraktion.

Zentrales Problem derartiger Kopplungs- und Adaptionskonzepte ist das fehlende Verständnis automatisierter Konzepte für die *Bedeutung* (Semantik) bestimmter Nachrichten und deren Abfolgen. Daher sollte Komponentenentwicklern die Möglichkeit geboten werden, entsprechend semantisch ausdrucksstärkere Spezifikationen anzugeben, die insbesondere jedoch auch durch automatische Verarbeitung dem System selbst zugänglich sein sollten. Ein wesentlicher Teil der vorliegenden Arbeit setzt sich mit dieser Problematik auseinander.

Eine gängige Art der Anpassung einer Komponente an eine aktuelle Einsatzsituation liegt in der Manipulation ihrer vom Komponentenwickler vorgesehenen Variationspunkte in Form von *Attributen*, bzw. auch als Eigenschaften oder englisch *Properties* bezeichnet, die sowohl zur Kompositions- wie auch zur Laufzeit verändert werden können. Dieses hier als *attributbasierte Anpassung* bezeichnete Vorgehen erhöht wiederum die Einsatzfähigkeit bzw. Flexibilität einer Komponente. Eine Anpassung an die Systemumgebung ist in diesem Rahmen somit trotz des Black-Box-Prinzips und der Philosophie vom »fertigen« Softwarebaustein in vom jeweiligen Komponentenentwickler gesetzten Grenzen möglich. Zur Vermeidung von Fehlkonfigurationen dieser Variationspunkte ist daher die Begleitung einer Komponente durch entsprechende Konfigurationswerkzeuge ein wesentliches Merkmal der Komponentensicht. Eine solche *angeleitete* Konfiguration trägt vorbeugend zur Fehlervermeidung bei. Im Rahmen der vorliegenden Arbeit ist daher großer Wert auf die Implemen-

tation entspechender Werkzeuge gelegt worden, die den vorgeschlagenen Prozess einer generativen Softwarekonstruktion begleiten.

Die Menge veränderbarer Attribute stellt dann eine *Konfigurationsschnittstelle* dar – im Gegensatz zu Programmschnittstellen, die die Grundlage der Komponenteninteraktion bilden. Selbstanpassende Systeme würden diese Grenze durch einen programmtechnischen Einfluss auf die eigenen Attribute verschwimmen lassen. Im Rahmen der gewünschten Automatisierung im hier vorgestellten Ansatz sollen auch entsprechende *automatische* Manipulationsmöglichkeiten zur konsistenten Konfiguration von Komponenten realisiert werden.

1.1.4 Verhältnis zur Objektorientierung

Konkret erweitert das Komponentenparadigma die auf klassischer Programmierung in 3GL-Manier[4] basierende Objektorientierung um neue Fragestellungen der *Kombination* und *Integration* abstrakter, gekapselter Softwarebausteine, die verglichen mit den vorhandenen Möglichkeiten der Verknüpfung von Klassen bzw. Objekten wesentlich »losere« Koppelungen – im Sinne einer Vermeidung wechselseitiger, insbesondere zusätzlich verdeckter Abhängigkeiten – ergibt. Dadurch stellen Komponenten gewissermaßen die »dinglicheren Objekte« dar. Andererseits sind es aber gerade auch die objektorientierten Ansätze, die durch Techniken wie etwa die späte Bindung (engl. *late binding*) zur Realisierung von Komponenten als am besten geeignet erscheinen. Vor allem die verteilten Objektsysteme wie etwa die *CORBA (Common Object Request Broker Architecture)* [OMG, 1996b] sind dabei geeignete Kandidaten, da ihnen von vornherein eine naturgemäß weitaus autonomere Objektsicht innewohnt.

Dabei ist die Komponentensicht bemüht, einige problematische Eigenschaften der Objektorientierung explizit zu adressieren oder bewusst auszugrenzen. Da ein Entwickler sich um die Implementierungsdetails einer Klasse nicht kümmern muss solange er (basierend zum Beispiel auf Erfahrungswerten) die Funktion einzelner Methodenaufrufe kennt und von der Fehlerfreiheit in den Implementierungsdetails ausgeht, konzentriert sich auch seine Arbeit eigentlich bereits auf das Zusammenspiel der verschiedenen Objekte eines Gesamtsystems. Dieses »Geflecht« ist jedoch nicht *eindeutig* und explizit, sondern eine inhärente Eigenschaft der resultierenden Gesamtanwendung. Erst

[4] Third Generation Languages

die Komponentensicht extrahiert und betont das Abhängigkeits- bzw. Nutzungsgeflecht explizit. In der Regel wird dies als Anwendungs*skelett* bezeichnet.

So erhöht das Konzept der *Vererbung* von Klassenimplementationen auf den ersten Blick zwar eine Wiederverwendbarkeit, führt aber zu einem *impliziten* Verhalten und verdeckten Abhängigkeiten der Softwarebausteine untereinander, die insbesondere bei erlaubter *Mehrfachvererbung* sehr schnell eine unüberschaubare Komplexität erreichen können.[5]

Eine weitere Schwäche tritt bei der Kopplung von Objekten auf. Der Anwendungsentwickler hat die für seine Applikation benötigten Klassen erstellt bzw. ausgewählt und legt durch die Spezifikation von Methodenaufrufen aufeinander Verbindungen zwischen den einzelnen Instanzen fest. Dann erfolgt in der Regel eine Transformation in eine maschinen-ausführbare Form durch einen Kompilationsvorgang. Die Laufzeitmodelle der meisten und vor allem gängigen Programmiersprachen legen damit die vorher spezifizierten Kopplungen zwischen den Objekten *unveränderlich* fest.

Das Szenario der zwei harmonisch zusammenspielenden Paradigmen ist jedoch auch anderweitig umstritten, denn gerade im Zusammenhang mit der Objektorientierung werden Softwarekomponenten oftmals nicht als eigenständiges Konzept, sondern lediglich als Synonyme angesehen. Es lässt sich festhalten, dass einerseits die Grenzen sicherlich – vor allem bei voller Ausschöpfung des breiten Spektrums an Objektsichten (zum Beispiel den oben angesprochenen »verteilten Objektsystemen« oder den »Business Objects« weiter unten) – verschwommen sind, andererseits aber ein Objektbegriff, wie er in der heutigen Programmierpraxis gängiger Produktionssprachen wie *C*++ üblich ist, den an Komponenten gestellten Anforderungen nicht entsprechen kann.

Programmiersprachen mit objektorientierten Konzepten wie C++ und Java dominieren die Praxis, garantieren für sich allein aber noch keine erfolgreiche Entwicklungspraxis, da das zugehörige methodische Vorgehen bei weitem keine so hohe Verbreitung bzw. Akzeptanz vorweisen kann. Ein großes Problem der Softwareentwicklung liegt zum Beispiel immer noch in der mangelnden Ausnutzbarkeit bestehender (Entwicklungs-) Ergebnisse und einer entsprechenden Adaptierbarkeit fertiger Softwarelösungen. Mechanismen wie etwa Vererbung und Polymorphis-

[5] Daher wird heute vielfach nach Möglichkeiten zur Entkoppelung der Vererbungshierarchien gesucht. [Smaragdakis, 1999] bietet etwa einen Ansatz mittels »Mixins«.

mus in der Objektorientierung tragen zu dem dringend benötigten Grad an Flexibilität bei, bergen jedoch auch eine neue Komplexität. Diese Komplexität sollte nicht manuell vom Entwickler gemeistert werden müssen, sondern durch entsprechende Werkzeuge handhabbar werden. Auch eine Komponentensicht wird genauso »scheitern« wie die Objektorientierung, wenn auch ihr es nicht gelingt, die zugehörigen Entwicklungsprozesse und Vorgehensmodelle in den Vordergrund des Entwicklerbewusstseins zu bringen.

Die Objektorientierung ist vom Grundgedanken der Möglichkeit, Modelle der Wirklichkeit im IT-System einfacher und *direkter* nachzubilden geleitet. So lassen sich etwa ein Aktenordner als Klasse modellieren und Methoden anbieten, um Dokumente abzuheften oder einzusehen. Diese Methoden entsprechen damit Tätigkeiten und die Klassen einem Gegenstand aus der realen Welt.

Der praktische Einsatz solcher objektorientierter Vorgehensweisen ist jedoch häufig durch eine 1 : 1-Abbildung zwischen Modell und Softwarestruktur geprägt, sodass Entwickler leicht Gefahr laufen, sich in der Problemanalyse und dem Lösungsentwurf zu sehr von technischen Gegebenheiten beeinflussen lassen und sich weniger auf die fachliche Ebene des Problems zu konzentrieren. Zugleich entstehen Softwarestrukturen, die zwar ein Bild der realen Welt widerspiegeln, dadurch jedoch nicht zwangsläufig auch zu optimalen Strukturen für die eigentliche Software führen, wenn Aspekte wie Wartbarkeit, Aktualisierbarkeit und Erweiterbarkeit betrachtet werden.

Die Komponentensicht versucht diese Problematik stärker zu berücksichtigen, indem zum einen ein grösserer Abstand zur technischen Realisierung geschaffen wird, zum anderen strukturelle Betrachtungen explizit neben die funktionalen Aspekte sowohl eines Modells als auch des fertigen Systems gestellt werden. Ersteres spiegelt sich etwa in der Betonung des Domänenverständnisses wider, letzteres zum Beispiel durch explizite Hilfsmittel wie Architektur-Beschreibungssprachen. Eine wesentliche Motivation der vorliegenden Arbeit stellt daher die Aussicht auf eine *modellgetriebene Anwendungskonstruktion* [Griffel, 2000a] dar, die die Eigenschaften des ursprünglichen Modellentwurfs verlustfrei in das realisierte System überträgt.

Eine analoge Entwicklung in der Objektorientierung selbst ist in der expliziten Einführung neuer Sprechweisen auszumachen, um die programmtechnische Ebene deutlicher von der konzeptuellen und modellhaften Betrachtung zu trennen. Ergebnis sind dann etwa Begriffe wie der des *Geschäftsobjektes* (engl.

Business Object) [Herzum & Sims, 2000]. Die Object Management Group (OMG) bemüht sich beispielsweise derzeit, solche Geschäftsobjekte für verschiedene Branchen zu standardisieren [Griffel, 1999b]. Die Business Objects sind dabei eigenständige Einheiten, die einen kompletten Gegenstand einer Anwendungsdomäne vollständig modellieren. Geschäftsobjekte werden in der Literatur auch häufig als Komponenten bezeichnet, da sie dem Komponentengedanken sehr nahe sind.

Business Objects sind Softwarebausteine, die in Hinblick auf Funktionalität und wechselseitiges Verhalten für komplette Kontexte von zum Teil umfangreichen Anwendungszenarien wie etwa betriebswirtschaftlichen Geschäftsprozessen stehen. Sie sollen dabei in ihrem nativen Zustand auf Anwenderebene gebrauchsfertig, d.h. erkennbar, ausführbar und kombinierbar sein. Dass *ein* Objekt im Sinne objektorientierter Programmierung für solche inhaltlich reichen Geschäftsobjekte auf Grund deren Komplexität ein wenig sinnvolles Implementationsmittel darstellt, ist offensichtlich. Der Komponentenansatz scheint hingegen auch hier ein gangbarer Weg zu sein. Kennzeichnend für Business Objects ist nach [Jacobsen et al., 1994] das Beinhalten eines »Controllers« und damit eben genau einer expliziten *Verhaltenssemantik*, wie es hier allgemein für einen Komponentenansatz gefordert wird.

Entwurfsmuster

Um die Erfahrungen, die bei der Lösung von häufig vorkommenden Problemen gemacht wurden, anderen Programmierern zukommen zu lassen und von den sich laufend verändernden technischen Details abstrahieren zu können, hat sich die Idee von Entwurfsmustern (engl. *Design Patterns*) durchgesetzt. In [Gamma et al., 1994] finden sich rezeptartige Lösungsvorschriften, die beschreiben, wie typische, sich wiederholende und komplexe Entwurfsprobleme mit Hilfe der Objektorientierung zu lösen sind. Entwurfmuster identifizieren, benennen und abstrahieren die Hauptaspekte wiederkehrender Strukturen und Verhaltensweisen einer Software. Beispiele für Muster sind etwa die *Factory*, die die Erzeugung neuer Objekte ermöglicht, oder der *Observer*, der die Überwachung und die Reaktion auf Veränderungen erlaubt.

Ein solches Muster, das zunächst nur ein konzeptionelles Konstrukt auf Papier bzw. eine Sprachkonvention unter Entwicklern darstellt, kann unter einer Komponentensicht auch ganz real werden. So wäre eben eine »Observer-Komponente« denk-

bar, die zum Beispiel Sensoren überwacht und durch Konfiguration auf verschiedenste Meßsituationen angepasst werden könnte. Umgekehrt eignen sich wiederkehrende Muster, die sich in der Realität finden lassen, dann auch zur Identifikation bzw. Spezifikation von Softwarekomponenten – stellen gewissermassen einen Bauplan dar. Ein typisches Beispiel ist etwa das Muster »Ressourcenplanung-auf-Zeitbasis«, das sich beispielsweise in der Reservierung von Hotelzimmern ebenso wie im Projektmanagement und vielen anderen Bereichen immer wieder findet. Eine Komponente, die die Problemstellung hinter diesem Muster generisch löst und sich dann durch Parametrisierung auf *bestimmte* Ressourcen oder *spezifische* Zeitformate einstellen lässt, stellt einen sinnvollen Basisbaustein dar, der in vielen Entwicklungsprojekten Verwendung finden kann. Insgesamt leistet das Konzept von Entwurfsmustern damit einen wertvollen Beitrag zu dem gewünschten verbesserten Übergang zwischen der Modell- und der Realisierungsebene, wenn es gelingt solche Muster direkt in Implementationscode zu überführen. Ein denkbarer Ansatz ist die Interpretation solcher Entwurfsmuster als »Mikro-Architekturen« als Grundlage zur Auswertung für eine automatische Code-Generierung [Batory & Geraci, 1995].

Frameworks

Enturfsmuster sind allgemeine Lösungsvorschriften für wiederkehrende Probleme in unterschiedlichen Anwendungen. Ein Rahmenwerk (engl. *framework*) ist dagegen eine Menge kooperierender Klassen, die unter Vorgabe eines Ablaufes eine generische Lösung für eine Reihe ähnlicher Aufgabenstellungen bereitstellen [Oestereich, 1999]. Sie fassen das Wissen über Funktionsweisen und architekturelle Strukturen eines Anwendungsbereiches (Domäne, s.a. Abschnitt 5.3) zusammen und kondensieren dieses in einer weiter- oder wiederverwendbaren Form. Im Unterschied zu Entwurfsmustern, die Wissen und Erfahrung über Software abstrakt repräsentieren, handelt es sich bei Frameworks in der traditionellen Sicht um Programmcode. Frameworks können als Materialisation eines oder mehrerer Entwurfsmuster angesehen werden [Griffel, 1998].

Ein Framework gibt damit eine Grundstruktur und Organisation der zu entwickelnden Software vor, die verfeinert werden muss. Bei einer Weiterentwicklung des Frameworks ist es jedoch nicht möglich, einzelne Bestandteile der Anwendung einfach durch neue, konkretisierte Versionen auszutauschen. Nach [Herzum & Sims, 2000] führt eine Framework-basierte Entwick-

lung immer noch zu monolithischen Anwendungen, d.h. es ändert sich zwar die Art *wie* entwickelt wird, nicht jedoch der *Charakter* der entstehenden Anwendungen. Hier bringt die Komponentensicht Vorteile, da sie vorgefertigte Bausteine voraussetzt, die für die Komposition zu Anwendungen vorgesehen sind, und eine Evolution durch ein einfaches Austauschen mit neuen Versionen von Einzelbausteinen leichter ermöglicht. Entsprechend schlagen [Batory & Smaragdakis, 2000] als Reaktion auf das Problem der Variation von abstrakten Klassen oder konkreten Instanzen, das den Erfolg von Rahmenwerken erschwert, die Dekomposition eines Frameworks in kombinierbare Komponenten vor.

1.1.5 Vorgehensmodelle und -prozesse

In der *visuellen Programmierung* werden Programmbausteine als Ikonen repräsentiert, die mittels grafischer Manipulationen – etwa dem wechselseitigen Verbinden durch Linienzüge als Abstraktion von Interaktionsbeziehungen – die Erstellung bzw. Modifikation eines aus ihnen zusammengesetzten Softwaresystems erlauben. Derartige Ansätze können ihr volles Potenzial am besten in Kombination mit Komponententechniken ausschöpfen, da solche als einzige alle notwendigen Eigenschaften, wie die einfache (Re-)Kombination bzw. Komposition zur Laufzeit und die flexible Anpassbarkeit durch ein angeleitetes Customizing bieten. Die zukünftige Bedeutung der visuellen Programmierung für große, langlebige Softwareprojekte wird in diesem Sinne stark von den Fortschritten im Bereich grundlegender Komponentenmodelle abhängen. Ein schönes Beispiel für visuelle Programmierumgebungen ist Suns experimentelles *Java Studio* Produkt [Sun, 1997b], das eine einfache »Verdrahtung« auf Basis von Komponenten mit hoher »Eigenintelligenz« erlaubt (vgl. Abb. 1.3).

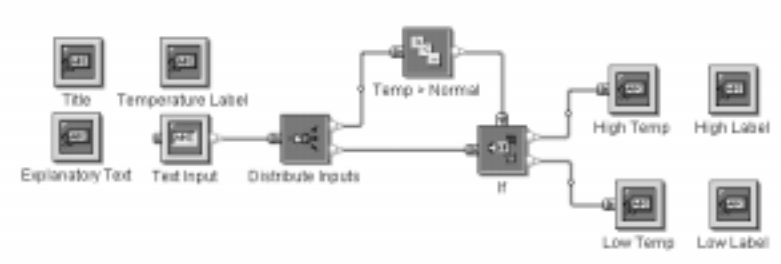


Abbildung 1.3
Visuelle Programmierung mit Java Studio-Beans

Ein gewisser Grad an Überprüfung einer »semantische Verträglichkeit« erfolgt in diesem Werkzeug automatisch, indem die einzelnen Komponenten über ein Transformationswissen bezüglich der auszutauschen Daten verfügen [Griffel, 1998]. Die meisten gängigen Entwicklungsumgebungen der Praxis berücksichtigen allerdings bislang solche Möglichkeiten nicht, da sie in der Regel mehr Entwicklungsaufwand auf Seiten des Komponentenentwicklers mit sich bringen. Das in dieser Arbeit vorgeschlagene *Typmanagement* erlaubt daher explizit die Unterstützung solcher Entwicklungsumgebungen, um bereits in der Konstruktionsphase entsprechende Überprüfungen zuzulassen, ohne das dies explizit im Code der eingesetzten Komponenten vorgesehen werden müsste.

Unified Modeling Language

Der Modellierungsphase wird in der Objektorientierung eine weitaus grössere Bedeutung beigemessen, als dies für viele frühere noch weit mehr technisch geleitete Methodiken der Fall war. Die Modellierung bietet durch entsprechende Abstraktionen und Vereinfachungen einen sinnvollen Ansatz, um die immer komplexer werdenden Systeme beherrsch- bzw. entwickelbar werden zu lassen. Ein wichtiges Hilfs- bzw. Ausdrucksmittel der Modellierung stellen dabei grafische Notationen dar. Es hat sich hierbei derzeit in der Entwicklungspraxis die Unified Modeling Language (UML) durchgesetzt (vgl. etwa [Oestereich, 1998]). Mit Hilfe entsprechender Werkzeuge können aus den UML-Modellen dann Teile der Implementierung, d.h. des Quellcodes generiert werden (sogenanntes *Forward-Engineering*). Dies betrifft besonders Programmteile, die in vielen Kontexten ähnlich ausfallen. Es können zum Beispiel Getter- und Setter-Methoden, Konstruktoren, Code zur Testunterstützung oder Persistenz automatisch aus dem Modell generiert werden. Es wird heutzutage aber nur 10-30% des Quellcodes automatisch erstellt, wobei sich die Codegenerierung hierbei hauptsächlich auf Klassendefinitionen beschränkt [Huber & Seidl, 1999].

Das sogenannte Roundtrip-Engineering wird bisher noch nicht vollständig unterstützt. Hierbei sollen Änderungen am Quellcode genauso automatisch in das Modell einfliessen, wie Änderungen am Modell neuen Quellcode ergeben. Für ein vollständiges Roundtrip-Engineering müsste das Modell alle Details, die für die Implementierung, nicht aber für die Modellierung eine Rolle spielen, enthalten und würde dabei unübersichtlicher werden.

Hier trifft man auf eine allgemeine Limitierung von visuellen Modellen: Je detaillierter oder komplexer die Modelle werden, desto unübersichtlicher werden sie. Ein Problem tritt beim Übergang von der Modellierung in die Implementierung auf. Das Designmodell sollte auf verständliche Weise ein Softwaresystem spezifizieren und von der Implementierung abstrahieren. Es erfolgt aber meist die schon erwähnte direkte $1:1$-Abbildung zum Beispiel der statischen Klassenstruktur eines objektorientierten Klassendiagramms. Ein anderer beachtenswerter Punkt grafischer Notationen stellt deren zwar für den visuell orientierten menschlichen Betrachter leicht verständliche, für ein Computersystem jedoch nicht direkt auswertbare Bildhaftigkeit dar. Entsprechende Werkzeuge müssen also zumindest intern andere Repräsentationsformen wählen und dabei notwendige Transformationen vor allem bruchlos und korrekt vornehmen, was aufgrund der mangelnden semantischen Stringenz der UML durchaus nicht trivial ist.

Geprägt von diesen Erfahrungen sollte der in dieser Arbeit angestrebte *fachnahe* Modellierunsansatz sich auf möglichst intuitve und überschaubare Darstellungsmittel beschränken, die nicht aus der technischen Sicht der objektorientierten Programmierung, sondern aus der problembezogenen Analyse des Einsatzbereiches der zu modellierenden Software abgeleitet wurden.

Vorgehensmodelle

Die UML bietet nur eine grafische Notation für die Softwareentwicklung. Wichtiger ist ein zugehöriges *Vorgehensmodell*, das die Herangehensweise und Organisation einer Softwareentwicklung anleitet und definiert. Es existieren mehrere solcher Entwicklungsprozesse. Beispiele sind etwa der Unified Process (RUP) der Firma Rational [Jacobson et al., 1999], das SEPP/OT-Rahmenwerk der Firma OWIS [Burkhardt, 1999] oder der komponentenbasierte Prozess Catalysis [D'Souza & Wills, 1998b]. Alle diese modernen Prozesse fordern eine architekturzentrierte Sicht, bieten diese zugleich aber zur Zeit jeweils nur mehr oder weniger. Grundsätzlich wird auch die *Domänenmodellierung* betont, wodurch der Fokus nicht nur auf der Entwicklung von jeweils einer einzigen Anwendung liegt. Zugleich wird der Endkunde in die Entwicklung direkt miteinbezogen (partizipative Entwicklung). Alle Prozesse sind auch explorativ, da der Endkunde während der laufenden Entwicklung bereits Prototypen erhält. Der Endkunde kann so seine Erfahrungen, die er bei der Verwendung der Prototypen gemacht hat, in den Entwicklungspro-

zess miteinfließen lassen. Die Änderungswünsche haben dabei eventuell Auswirkungen auf das anfangs zu Grunde gelegte Modell des Systems. Auf diese Weise entsteht also ein *evolutionärer* Prozess mit iterativen und zyklischen Phasen. Prozesse wie der RUP betonen leider oft noch die Phasensicht zu unabhängig vom eigentlichen Gegenstand der Entwicklung, machen also Phasenübergänge nicht direkt am Entwicklungsstand der Zwischenergebnisse fest [Hesse, 2000].

Beispiel: Catalysis *Catalysis* ist derzeit einer der interessantesten weil vollständigsten und komponentenbasierten Modellierungsprozesse, der primär die Entwurfs- und Entwicklungsphase der Anwendungserstellung durch Modellierungstechniken, die auf dem bestehenden Modellierungskonzept der UML aufbauen, erleichtern soll. Die Anwendung wird dabei in einer Designphase zunächst auf einem hohen Abstraktionslevel entworfen. Mittels *Refinement* (Verfeinerung) werden diese Entwürfe später exakt spezifiziert.

Anwendungsentwicklung wird dabei als ein Zusammenfügen vorgefertigter Bausteine (Komponenten) verstanden. Für die benötigte möglichst universelle Kopplungsfähigkeit einzelner Komponenten fordern [D'Souza & Wills, 1998a] eine möglichst kleine Anzahl *standardisierter* Interfaces (Schnittstellen) verglichen mit der Anzahl verwendeter Komponenten. Diesem Ziel dient die Konstruktion von »Konnektoren« sowie präziser Schnittstellenspezifikationen, die mittels der Angabe von Vor- und Nachbedingungen und des »Ziels« einer Interaktion Informationen über die Bedeutung und die verdeckten Abhängigkeiten verwendeter Methodenaufrufe an eine oder ausgehend von einer Komponente enthalten. Ein solcher Konnektor legt dann wiederum einen bestimmten *Interaktionstyp* fest, der dem oben erwähnten Begriff des Interaktionsmusters entspricht.

Ein derartiger Entwurfsprozess eignet sich hervorragend für die Design- und Kompositionsphase komponentenbasierter Anwendungen, da Interaktionen auf beliebig hohen Abstraktionsebenen definiert werden können. Auf diesen Ebenen wird den internen Arbeitsabläufen der Komponenten und den (technischen) Details des benötigten Interaktionsprotokolls keinerlei Beachtung geschenkt.

Konnektoren sollen in der Kompositionsphase ebenso als »Black-Boxes« betrachtet werden, wie andere beteiligte Komponenten. Die Kopplung zweier Komponenten wird durch das Einfügen eines gewünschten Konnektors realisiert. Die mit dem Einsatz eines Konnektors einhergehende *Typisierung* der Interakti-

on verringert die Gefahr einer semantisch inkompatiblen Kopplung. Zudem ermöglicht die Typisierung eine formale Nachprüfbarkeit potenzieller Kopplungen. Um diese theoretisch postulierten Vorteile in der Entwicklungspraxis umsetzen zu können, fehlt es derzeit an entsprechend konkreter Systemunterstützung für den Catalysis-Ansatz, da auch hier wieder keine Hinweise auf einen direkten und verlustfreien Übergang in die Implementation eines Systems gegeben werden. Es stellt sich insbesondere auch die Frage, wie das »Innenleben« der Konnektoren realisiert werden kann und wie sich die semantisch reichen Informationen des Modells in der Implementation wiederfinden sollen. Diese Fragestellungen haben wiederum zur Motivation der Untersuchungen adaptiver und klassifizierender Konzepte im Rahmen der vorliegenden Arbeit beigetragen.

eXtreme Programming Ein gutes Kennzeichen dafür, wie wenig Entwickler aus der Implementationspraxis bereit sind, entsprechende Vorgehensmodelle und Modellierungstechniken zu verinnerlichen, zeigt das aktuell große Interesse am von Kent Beck vorgeschlagenen *Extreme Programming* (kurz XP) [Josuttis, 1999]. Der heute typische Entwickler sieht üblicherweise immer noch die Codierung als den Schwerpunkt seiner Arbeit. Daher ist die XP-Methodik bestrebt, nicht das Modellieren, sondern das Programmieren (wieder?) in den Vordergrund zu stellen [Jeffries, 1999]. Das Modell entsteht dabei bestenfalls erst beim Programmieren selber. XP stellt damit einen Gegentrend zu den oben genannten modernen Entwicklungsprozessen dar.

Die wichtigsten Aspekte des *eXtreme Programming* sind das *Testen* und das Codieren. Das Gesamtproblem wird in kleine Entwicklungsschritte faktorisiert. Es werden *zuerst* Tests geschrieben, die das Verhalten des Systems *operational spezifizieren* sollen. Erst dann wird der eigentliche Zielcode entwickelt. Am Ende eines Entwicklungstages soll immer ein lauffähiges System vorhanden sein. Das System wird entsprechend refaktorisiert, wenn dies nicht gegeben sein sollte. Die Tests erlauben eine kontinuierliche Verifikation und Validierung des Systemverhaltens. Allerdings müsste diskutiert werden, in welchem Verhältnis die Programmierung von Tests zur eigentlichen Systemerstellung steht, was sowohl den Codeumfang als auch die Entwicklungszeiten und insbesondere wiederum die Fehlerfreiheit *der Tests* angeht.

Es wird immer im Team von 2 Leuten programmiert, wobei einer konkret am Computer den Code formuliert und der andere eine mehr strategische Sicht auf das System hat und den Code

kontrollieren kann. Die Rollen sollen hierbei zwischendurch getauscht werden. Der entwickelte Code ist kollektives Eigentum und nicht einem Programmierer alleine zuzuordnen. Dies soll zu einer höheren Qualität der Software führen und die Abhängigkeit zu einzelnen Programmierern und ihrer subjektiven Denk- und Vorgehensweise mindern.

Die Stärke dieser Vorgehensweise liegt auf Grund der geforderten Teamkooperation und -kommunikation in kleineren Teams von 6 bis 12 Entwicklern. Damit bietet sich XP insbesondere für die Implementierung von abgrenzbaren Softwareeinheiten wie eben Komponenten an. Die Kombination mit der Komponentensicht bietet auch den Vorteil, dass die von XP betonten Unit-Tests unabhängig von der eigentlichen Anwendungserstellung stattfinden – eben in der Entwicklung *von* Komponenten – und so besser die Argumentation für solche frühen Tests gerechtfertigt werden kann, da die spätere Anwendungserstellung nicht mehr von ihnen unterbrochen wird.

Der neben der reinen Euphorie durchaus nachweisbare Erfolg (in Form rasch den Kunden befriedigender Lösungen) mit ebenso nachweisbaren Misserfolgen (etwa auf Grund fehlender Architekturübersichten oder Dokumentationen) dieses »extremen« Ansatzes führte zu der Grundannahme der vorliegenden Arbeit, dass eine Modellierung zwar unabdingbar den Mittelpunkt einer Softwareentwicklung prägen sollte, diese jedoch simpel, intuitiv und entsprechend systemtechnisch unterstützt sein sollte, um die Akzeptanzschwelle eines Anwendungsentwicklers zu unterschreiten. Gleichzeitig scheint der Lösungsansatz dieses Akzeptanzproblems in der Trennung der Entwickler in die von der Komponentensicht geforderten *Komponentenentwickler* und die *Anwendungskonstrukteure* zu liegen, da für letztere eine ausreichende Abstraktion von den eingesetzten Technologien erreicht werden kann, wie die vorliegende Arbeit zeigt.[6]

Komponentenorientierte Anwendungsentwicklung

Grundidee des hier vertretenen Komponentenparadigmas ist die Softwareentwicklung durch das Zusammensetzen (die Komposition) *vorgefertigter*, separater Teilbausteine (Komponenten). Dieser Vorgang wird zur Abgrenzung von der üblichen Softwareentwicklung – insbesondere grundlegender Komponenten selbst –

[6] Ein generativer Ansatz bietet sich sowohl zur Erstellung von Einzelkomponenten als auch für eine Automatisierung der Anwendungsentwicklung selbst an. In dieser Arbeit steht aber die Anwendungskonstruktion im Vordergrund.

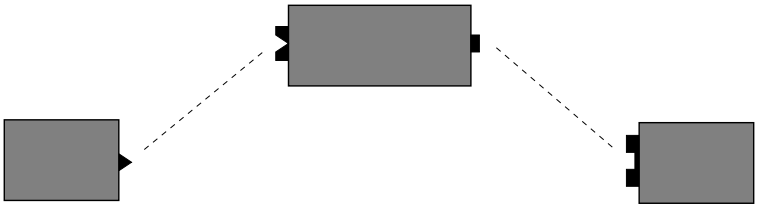


Abbildung 1.4 *Anwendungserstellung durch Komposition mittels Komponentenkopplung festgelegter Schnittstellen*

als Software*konstruktion* bezeichnet. Grundsätzlich lassen sich folgende Phasen einer komponentenbasierten Softwareentwicklung unterscheiden:

- In der *Entwicklungsphase* werden benötigte Komponenten bzw. Teilbausteine entwickelt.
- In der *Kompositionsphase* werden Komponenten zu einer Gesamtanwendung zusammengefügt.
- In der *Laufphase* läuft die Anwendung. Veränderungen während der Laufphase werden als *dynamische* Veränderungen bezeichnet.

Benötigte Teilbausteine können intern wiederum aus mehreren Komponenten zusammengesetzt sein. Für die Kompositionsphase stellt sich ferner die Frage nach einer *Klassifikation* vorhandener Bausteine bzw. deren Auffinden. Gegenstand dieser Arbeit ist daher insbesondere auch die Untersuchung der Implikationen entsprechender *Komponentenmodelle* sowie die Erarbeitung geeigneter Klassifikationskonzepte. Die Komponentensicht fördert dabei die Idee, dass die (Selbst-) Beschreibung ein integraler Bestandteil der Komponenten selbst ist.

Zur Bedeutung der Architektur

Die oben angesprochenen Entwicklungsprozesse wurden als »architekturzentriert« bezeichnet, d.h. der Begriff der *Architektur* rückt in den Vordergrund des Entwurfsprozesses und verbessert das Systemverständnis auf fachlicher und struktureller Ebene mit frühzeitigen Aussagen und Analysen, etwa zu Qualität und Robustheit [Klein et al., 1998] [Barbacci et al., 1998]. Dies geht soweit, dass etwa [Bass & Kazman, 1999] eine architekturbasierte Entwicklung als eigenes Paradigma fordern (vgl. a. [Woods et al., 1999]). Obwohl vermutlich niemals ein einzelner

Standard die Softwareindustrie dominieren wird, kann doch eine (automatische) Restriktion auf einen Basissatz von Architekturstilen beobachtet werden [Garlan, 1995] – letztlich entspricht auch die Idee der Entwurfsmuster dieser Denkweise. [Kazman & Carriére, 1997] diskutieren anschaulich wie dabei die Vergröberung einer Systemsicht zur Extraktion *überschaubarer* Architekturen führen kann. Dies entspricht im Gegensatz üblicher objektorientierter Strukturen auch dem Komponentengedanken mit *wenigen* Verknüpfungen im System. Überhaupt kommt die Komponentensicht dem architekturellen Denken sehr entgegen, da sie Architekturinformationen explizit werden lässt und so zugleich das bisherige oftmalige Fehlen entsprechender Architekturdokumentation adressiert. Ein generativer Ansatz stützt dies nochmals, da er direkt das Vorliegen expliziter Architekturmodelle fordert, sodass eine weitergehende Formalisierung solcher Modelle vorgenommen werden muss. Werden Architekturen dabei als charakteristisch für jeweils betrachtete Anwendungsbereiche (Domänen) erkannt, hilft die frühzeitige Konzentration auf solche domänenspezifischen Merkmale die durch den »bottom-up-«-Charakter der Komponentensicht drohende Gefahr der Vermischung von Analyse- und Designphase zu mindern.[7] Umgekehrt bieten die Verknüpfungsmöglichkeiten von vorhandenen Komponenten natürlich durchaus auch Hinweise auf denkbare Architekturen.

Der hier vorgestellte generative Ansatz ist von der Nutzung vorgegebener Architekturen als »Baupläne« inspiriert, die eine automatische Ableitung mehrerer, einer *gemeinsamen* Architektur entsprechenden konkreten Applikationen im Sinne einer System*familie* zulässt.

1.1.6 Charakterisierung des Komponentenbegriffs

Im Hinblick auf den Komponentenbegriff hat die Vergangenheit eine Vielzahl von Definitionen hervorgebracht. Als wesentlich hat sich dabei der *bausteinartige* Charakter sowie die Kapselung interner Implementationsdetails herausgestellt, sodass die Softwarekomponente als eine Art geschlossenes Softwarekonstrukt (»Black-Box«) aufgefasst werden kann, das über wohldefinierte Verknüpfungspunkte (Schnittstellen – passiv, Konnektoren – ak-

[7] Das Vorhandensein fertiger Bausteine kann den Entwickler verleiten, für eine Problemlösung nur die Möglichkeiten der vorhandenen Komponenten zu betrachten, statt zunächst eine (optimale) Analyse des Problems vorzunehmen und dann erst eine *Zuordnung* zu vorhandenen Teillösungen vorzunehmen.

tiv) potenzielle Kollaborationsmöglichkeiten mit anderen Komponenten spezifiziert und damit die *angebotenen* sowie die *benötigten* Funktionalitäten festlegt.

Abhängig von der jeweiligen Blickrichtung verschiedener Ansatzpunkte findet sich diese Komponentenidee in vielfältigen Ausprägungen wieder. Aus der Sicht verteilter Systeme steht der Interaktionscharakter separater Einheiten im Vordergrund. Unter softwaretechnischen Aspekten ist vor allem die Vorgehensweise der Anwendungskomposition innerhalb von Rahmenwerken und Mustern interessant. Forschungsbereiche, die sich mit der Wiederverwendbarkeit von Software-Artefakten beschäftigen, haben den Komponentenbegriff traditionell in sehr breiter, allgemeiner Weise geprägt und zielen dabei auf Entwicklungsstrategien bzw. Vorgangsmodelle ab. Das Umfeld des »programing in the large« [Griffel, 1998] ist unter anderem mit Koordinationssprachen besonders an dem »Klebstoff« (engl. *glue*) interessiert, der die Komponenten verbindet und zusammenhält. Insgesamt zeigt diese Weiterentwicklung in der Sichtweise damit ein Hervorheben größerer Strukturen im Sinne des Fokus auf den oben diskutierten Begriff der *Architektur*.

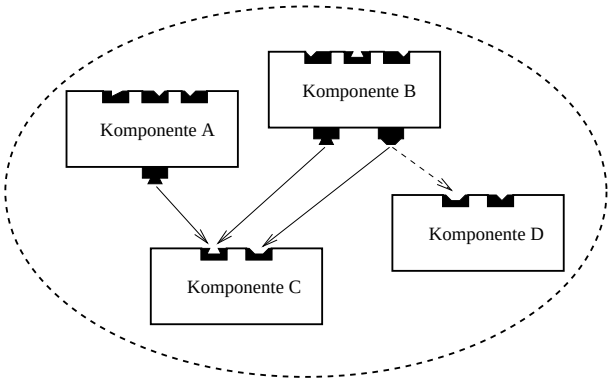


Abbildung 1.5
Softwarekomponenten

Die verschiedenen Sichten führen jedoch zu einer großen Bandbreite möglicher Auslegungen, sodass der Komponentenbegriff nicht leicht zu fassen ist. Daher soll seine Eingrenzung im Folgenden durch die Gegenüberstellung bislang in der Literatur zu findender Begriffbestimmungen erfolgen.

Der auf bestehenden Artefakten basierende Komponentenbegriff kann zunächst sehr generisch gehalten werden, ohne auf spezifische Details einzugehen. Holibaugh und Perry definieren zum Beispiel Komponenten als

»*... logical part of a system or program.*«

[Holibough & Perry, 1988]. Eine solche Definition ist noch deutlich von der klassischen Modulsicht der Programmierung geprägt und gibt dem Komponentenbegriff wenig eigenständigen Charakter. Ebenso allgemein spricht Kain von Komponenten als
»*... a product of the development process that exhibits certain qualitys of usability and separability.*«
[Kain, 1996] und nach McGregor et al. gilt:
»*A component is any unit that provides a relatively independent piece that is used in combination with a number of components in different configurations.*
[McGregor et al., 1996]. Dies entspricht dann bereits weit deutlicher einer von programmiertechnischen Details abstrahierenden modernen Komponentensicht.

Hooper und Chester identifizieren den Begriff der Komponente sehr stark mit dem der Wiederverwendung, denn sie nutzen ihn
»*... to mean any type of software resources that may be reused (e.g., code modules, designs, requirements specification, domain knowledge, development experience, or documentation)*«
[Hooper & Chester, 1991]. Ganz ähnlich umfasst auch der Ansatz innerhalb des NATO-Standards zur Entwicklung wiederverwendbarer Softwarekomponenten jegliche Artefakte: Eine Software-Komponente ist
»*... a software entity intended for reuse,...* « , die
»*... may be design, code or other product of the software development process.*«
[Braun, 1994]). Zwar schon konkreter auf einen technischen Entwicklungsprozess bezogen und die »Verpackungseinheit« einer Komponente betonend, aber immer noch sehr allgemeingültig sagt D'Souza:
»*A component can include anything a* **package** *can include: executable code, source code, designs, specifications, tests, documentation, and so on.*«

Derartige generalistische Definitionsansätze vermögen nur eine diffuse Vorstellung von Komponenten zu vermitteln und taugen damit nicht wirklich, ein Komponentenparadigma in der praktischen Softwareentwicklung etablieren zu können. Andere Definitionen lassen konkretere, konzeptionelle Aspekte einfließen und sind daher anschaulicher und zielen entsprechend mehr auf tatsächlich ausführbare Programmeinheiten ab.

Nach Nierstrasz sind Komponenten
»*static abstractions with plugs*«
[Nierstrasz & Dami, 1995]. Eine Komponente ist danach zunächst eine abstrahierende Kapsel die zudem mit den *Plugs* defi-

nierte Zugangspunkte besitzt. Der problematischste Aspekt dieser Definition ist wohl die Eigenschaft der *Statik*, die an den Klassenbegriff der Objektorientierung erinnert und damit dem instanzlichen (»dinglichen«) Charakter einer Komponente nicht gerecht wird.

Von Szyperski und anderen stammt folgende Sicht:
»*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be developed independently and is subject to composition by third parties.*
[Mühlhäuser, 1997]. Die ausschließlich expliziten Abhängigkeiten und die unabhängige Entwicklung stehen hier im Vordergrund. Wegner sieht die Komponentensicht als Nachfolger der Objektorientierung und charakterisiert Komponenten plastisch als
»*. . . a generalization of objects that extends the primitives for realizing interaction to include distributed components, graphical user interfaces, databases, robots and virtual reality.*«
[Wegner, 1993], wobei hier die Interaktionen in den Mittelpunkt des Interesses rücken. Die folgenden Definitionen (vgl. a. [Griffel, 2001]) spiegeln nach Meinung des Autors die wesentlichen – entsprechend hervorgehobenen – Attribute des Komponentenbegriffs aus unterschiedlichen Blickwinkeln am besten wider und haben die Komponentensicht der vorliegenden Arbeit entsprechend inspiriert:
Booch: »*A reusable software component is a* **logically cohesive**, **loosely coupled** *module that denotes a single abstraction.*«
Jacobson: »*Components can be regarded as* **completed** *building elements which are already* **placed in the implementation space** *and that can be used directly in our system.*«
Meta Group: »*Software components are defined as* **prefabricated, pretested, selfcontained, reusable** *software modules – bundles of data and procedures – that perform specific functions.*«
Sims: »*A component is built for* **composition and collaboration** *with other components.*«
OMG: »*Business Components are designed to support a clearly defined* **relationship** *between business* **processes** *and software* **implementation**.«

Weitere Lieferanten von Komponentendefinitionen sind die vielfältigen kommerziellen Systeme mit unmittel- oder mittelbarem Komponentenbezug, zu denen etwa Suns *Java Beans* oder Microsofts COM+ zu zählen sind. Solche technisch konkreten Beispiele wurden in [Griffel, 1998] ausführlicher diskutiert und werden im Rahmen dieser Arbeit nicht näher betrachtet.

Mit steigendem Maß an Bestimmtheit werden Komponenten dabei entsprechend vielfach als spezifische Sprachkonzepte verstanden. Hierbei muss beachtet werden, dass das Potenzial des Paradigmas dann nur zum Teil ausgeschöpft werden kann, da sie gewissermaßen einen »Rückschritt« in Richtung programmtechnischer Konstrukte darstellen. Wird dies jedoch bewusst einkalkuliert, ergeben sich erstmals direkt technische Umsetzungen der genannten Ansätze.

Insgesamt ergibt sich also ein Bild fertiger, d.h. implementationsseitig definierter, getesteter Bausteine, deren voneinander unabhängige Entwicklung mit explizitem Hinblick auf ihre spätere Zusammenarbeit erfolgt, wobei aber eine logische Nähe durch die Einordnung in gemeinsame Einsatzbereiche bzw. eine Zuordnung zu definierten Prozessen einer Domäne berücksichtigt wird. Definitionen wie die D'Souzas sind allerdings wenig hilfreich, da sie derart generisch mit dem Begriff der Komponente umgehen, das die Einführung des Begriffes keinen Gewinn erzielt, da die unter ihm zusammengefassten Artefakte gar nicht gemeinsam und homogen verarbeitet bzw. beschrieben werden können. Die Hauptströmung der aktuellen Komponentendiskussion bevorzugt daher die angesprochene implementationsbetonte (»physische«) Sicht.

Grundlage der vorliegenden Arbeit ist im Folgenden ein Komponentenbegriff, der von der Granularität des korrespondierenden Softwarebausteins abstrahiert und vielmehr dessen *Kopplungs*- und *Interaktionsfähigkeit* in den Vordergrund stellt. Komponenten befinden sich zudem stets in einem ausführbaren Zustand, sodass sie vor ihrem Einsatz aus einem entsprechenden Repository entnommen werden können.

Komponenten werden dabei aus Modellsicht jedoch weiter als autonome Abstraktionseinheiten betrachtet. Sie sollen sich an eigenständig ausführbaren Software- bzw. Code-Einheiten orientieren, dabei aber nicht auf bestimmte Konzepte – wie objektorientierte oder funktionale Paradigmen – einschränken, weshalb die Definition auch auf einem relativ abstrakten Niveau erfolgt und zunächst keine Einzelheiten der technischen Umsetzung enthält. Auf methodologischer Ebene soll den Komponenten ebenso eine semantische Abgeschlossenheit wie die fundamentale Intention zu zielgerichteter, wechselseitiger Kooperation innewohnen.

Definition 1.1 *Eine* Komponente *ist die softwaretechnische Abstraktion eines domänenspezifischen Fachkonzepts, das mindestens eine semantisch geschlossene* Interaktion *im Sinne des voll-*

ständigen Austausches zugehöriger Domänenaktivitäten *mit einer anderen Komponente ermöglicht.*

Im Verlauf der folgenden Kapitel werden die hier verwendeteten Begriffe ihrerseits näher bestimmt, sodass diese Definition hier zunächst nur zur ersten Illustration des Standpunkts des Autors dienen soll und sich erst später vollständig erschliessen wird. Eine Komponente wird dabei als unabhängig implementierte Kapsel mit vollständig spezifizierter Oberfläche und Verhalten betrachtet, wobei die dazu verwendeten Implementationstechniken durch die Einführung eines entsprechenden *Komponententyps* zum einen verborgen, zum anderen aber auch in Bezug zu den genannten fachlichen Konzepten gesetzt werden.

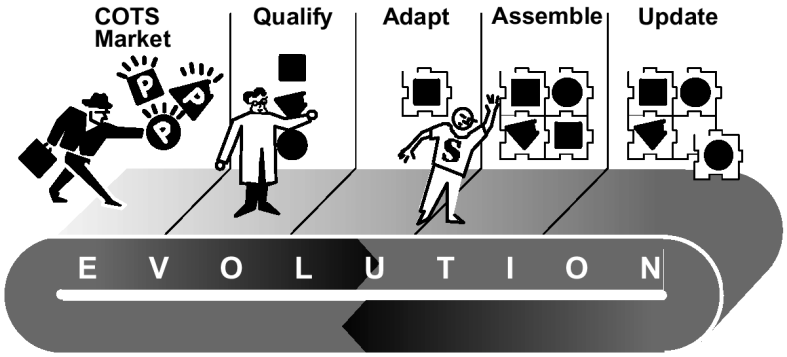
1.2 Zielsetzung und Gang der Untersuchung

Die vorliegende Arbeit hat sich in der Tradition von [Griffel, 1996] den Nachweis der Praktikabilität eines durchgängig von implementatorischen Details abstrahierenden und vollständig auf der Verwendung vorgefertigter Softwarebausteine basierenden Entwicklungsprozesses zum Ziel gesetzt. Ziel des Entwicklungsprozesses ist dabei die Unterstützung von Fachleuten eines Problembereichs derart, dass diese selbstständig eine vollständige Softwareapplikation unter alleiniger Ausnutzung ihrer eigenen Fachkenntnisse erstellen können. Die für diesen Nachweis notwendigen Implementationsarbeiten bedingen jedoch zunächst entsprechende konzeptionelle Vorarbeiten, die im Fokus der vorliegenden Darstellung stehen. Bei diesen wiederum liegt der Schwerpunkt gemäß der Anforderungen der allgemeinen Zielsetzung der Nutzung bestehender Funktionalitäten auch im Falle technischer Heterogenität oder unterschiedlichem Verhalten zur Erbringung gleicher Funktionen auf Fragestellungen der *Adaption* und *Klassifikation* von Softwarekomponenten. Die gewünschte Entlastung des Softwarekonstrukteurs von IT-technischen Details erfordert dabei eine weitgehend *automatisierte* Applikationserstellung. Ziel ist daher auch die Bereitstellung eines entsprechenden *Generators*. Für den gesamten Konstruktionsprozess steht als primäres Ziel im Vordergrund, im Rahmen einer Modellierung festgelegte bzw. identifizierte Randbedingungen und Eigenschaften im implementierten System verlustfrei umzusetzen bzw. aufrecht

zu erhalten. Dies erfordert eine zwar fachnahe, aber dennoch direkt technisch auswertbare Klassifikation von Bausteinen.

Die Arbeit adressiert damit die bestehende Lücke zwischen umfangreichen Überlegungen zu Architekturen und einem zugehörigen Domain Engineering auf der einen Seite und der *konkreten spezifischen Systementwicklung* auf der anderen. Die [Oberndorf et al., 1997] entnomme Abbildung 1.6[8]illustriert dabei sehr schön die Sichtweise, von der diese Arbeit ausgeht. Ziel ist es entsprechend die mittleren drei Phasen so weit wie sinnvoll möglich zu automatisieren und die vierte Phase durch entsprechende Mechanismen und Konzepte zu unterstützen.

Abbildung 1.6
Der Baustoffzyklus



Eine solche automatisierte Systemkomposition soll dabei angeleitet und *konsistenzerhaltend* angelegt sein (vgl. a. [Batory et al., 1999a]), d.h. es soll zum einen verhindert werden, dass Komponenten eingesetzt werden, deren Kombination semantisch unsinnig ist, zum anderen eine konsistente Konfiguration der Gesamtkomposition erreicht werden, indem gewisse *systemdurchdringende* Eigenschaften automatisch erzwungen werden.

Ziel dieser Arbeit ist damit zum einen ein Erkenntnisgewinn bezüglich der komponentenbasierten Sicht, ihrer Implikationen und Konzepte, zum anderen die Ausarbeitung und Realisierung konkreter Mechanismen zur Demonstration des Potenzials und des praktischen Nutzens einer komponentenzentrierten Softwareentwicklung. Das Leitbild der Betrachtungen bildet dabei das hier vorgeschlagene Konzept einer *generativen Softwarekonstruktion* mit der gegenüber anderen Ansätzen konsequenten Verlagerung der Entwicklungssicht von der technologischen

[8]Hier wird der Zyklus eines Massenmarktes von *Commercial-Of-The-Shelf* Produkten zu Grunde gelegt.

Programmierebene hin zu anwendungsbezogenen Problemlösungen. Dieser Ansatz geht vom Vorhandensein fertiger Teilfunktionalitäten in Form von Komponenten aus und vermeidet so viele Detailprobleme der Quellcode-Manipulation und -erstellung, beispielsweise ein komplexes Round-Trip-Engineering unter möglichem Verlust nachträglicher Ergänzungen durch den Entwickler.

Andererseits wirft ein solcher weitgehend automatisierter Generator-basierender Ansatz eine Anzahl spezifischer Probleme auf, die es zu lösen gilt. So stellt sich die Frage nach einer geeigneten Darstellung architekturellen Wissens bzw. der Erarbeitung eines solchen Wissen. Letzteres erfolgt im Rahmen eines Domain Engineerings, das jedoch neben einem kurzen Exkurs nicht Gegenstand dieser Arbeit ist. Vielmehr setzt der vorgestellte Ansatz am Punkt der Wissensdarstellung ein, d.h. der notwendigen *Formalisierung* des erarbeiteten Wissens über ein spezifisches Anwendungsumfeld (Domäne).

Grundidee des Ansatzes ist das Vorlegen einer Architekturspezifikation für jeweils eine bestimmte *Produktlinie* innerhalb einer Domäne und das teil-automatisierte »Ausfüllen« eines solchen architektonischen Rahmens mit konkreten Funktionsbausteinen. Daraus ergibt sich als erste wesentliche Fragestellung, wie die Repräsentation der Architektur mit der Repräsentation der auszuwählenden Einzelkomponenten in Wechselwirkung zu bringen ist und wie dies die Arbeitsphase eines Generators beeinflusst.

Hieraus leitet sich unmittelbar die zweite wesentliche Fragestellung ab, nämlich nach der Art der Beschreibung der Bausteine. Eine solche Beschreibung muss einerseits eine Auswahlentscheidung auf Anwendungsebene durch den Generator (eventuell auch Entwickler) erlauben, andererseits genügend Information für die anschließend vom Generator durchzuführende technische Kopplung der Komponenten enthalten. Darüber hinaus sollte sie auch in der Lage sein, ausreichende Informationen für die manuelle Konfiguration als Zwischenphase des Entwicklungsvorgangs bereitzustellen. Solche Komponentenbeschreibungen werfen ihrerseits wiederum die Frage nach dem eigentlich zu Grunde zu legenden Komponenten*modell* auf.

Gemäß diesen Fragestellungen beziehungsweise deren jeweiligen Teilproblemen gliedert sich der Gang der Untersuchung und damit die Darstellung der nachfolgenden Kapitel in entsprechend umgekehrter Weise.

Als Grundlage der Diskussion hat diese Einleitung zunächst mit der Vorstellung und Einordnung aktueller Aspekte der Komponentendiskussion begonnen, mit der Betonung einer interakti-

onsorientierten, konnektionistischen Sichtweise. Darauf aufbauend diskutiert Kapitel 2 formal abstrakte Aspekte eines Komponentenmodells als Ausgangsbasis der Betrachtung kompositorischer Anforderungen. Wesentlich für das Modell ist dabei sein azyklischer, hierarchischer Charakter zur Minimierung gegenseitiger Abhängigkeiten innerhalb von Softwarestrukturen. Die dabei untersuchten logikbasierten Semantikrepräsentationen lassen die Ergänzung um ein einfacheres operationales Verhaltensmodell auf Basis auszutauschender Nachrichten als sinnvoll erscheinen, das in Kapitel 3 erörtert und zur Ableitung adaptierender Mechanismen genutzt wird, um die Menge vorgefertigter Bausteine bei Bedarf besser ausnutzen zu können.

Unter Verwendung dieser operationalen Spezifikationen wird im Anschluss in Kapitel 4 zu diesem Komponentenmodell ein Typmodell mit einem zugehörigen Verarbeitungsmechanismus (Typmanager) entworfen und entsprechende Kategorisierungs- und Beschreibungsmöglichkeiten für Einzelbausteine einer Anwendung vorgestellt. Der Fokus liegt auf der Sichtweise, dass Komponenten zwar möglichst weitgehend selbst alle zu ihrem »Einbau« notwendigen Informationen liefern, diese Informationen aber unabhängig von der Realisierung der eigentlichen Funktionalität einer Komponente vorliegen sollten, um den Grad der Wiederverwendbarkeit zu erhöhen.

Ausgehend von diesen Klassifikationsmöglichkeiten wird dann in Kapitel 5 ein architekturgetriebenes Generatorkonzept entwickelt, dass eine konsistente, weitgehend automatisierte Konstruktion auch potenziell verteilter Softwareapplikationen erlaubt. Grundgedanke ist hierbei die selbstständige Suche und Auswahl zu einem vorgegebenen Architekturplan »passender« Komponenten durch einen Generator, der auch die Verknüpfung der Einzelbausteine durchführt. Der Anwendungsentwickler sollte dabei nur noch im Rahmen zulässiger Konfigurationsoptionen manuell eingreifen müssen, indem er individuelle Komponenteneigenschaften oder gewünschte Verhaltensweisen aus dem möglichen Spektrum auswählt. Es handelt sich also um die gewünschte »angeleitete« Softwareentwicklung, die unter gleichzeitiger (automatischer) Auswertung von Konstruktionsrichtlinien erfolgt, um eine konsistente Anwendungserstellung zu erlauben. Schwerpunkt dieser Vorgehensweise ist dabei ein domänenzentrierter Blickwinkel, der auf die Betrachtung bzw. Bearbeitung programmiertechnischer Fragestellungen vollständig verzichtet. Stattdessen erhält der Anwendungsentwickler die Möglichkeit ausschließlich in (fachlichen) Dimensionen und Begriffen des Problemraums zu denken und zu handeln. Daher wird ne-

ben einem Vergleich verwandter Ansätze auch ein entsprechender Exkurs des Domain Engineering vorgenommen.

Kapitel 6 beschreibt abschließend die konkreten Implementierungen der zuvor eingeführten Konzepte und Mechanismen und erläutert dabei eingesetzte bzw. entworfene Algorithmen. Kapitel 7 zieht ein Resümee der diskutierten Aspekte und Ergebnisse. Eine Reihe von Anhängen liefert Detailinformationen zu den Betrachtungen der einzelnen Kapitel, insbesondere verwendete Formalismen und Illustrationsbeispiele werden gezeigt.

Zwecks einer geschlosseneren Darstellung der Gesamtarbeit wird insgesamt keine vollständige Trennung zwischen den Konzeptions- und den Implementationsbereichen vorgenommen, d.h. einige technologische Entscheidungen werden bereits im Rahmen der Konzeption beschrieben und einige konzeptionelle Diskussionen finden sich im Implementationskapitel.

2 Modell

Zur Festlegung eines Komponentenmodells[1] und der mit ihm verbundenen architekturellen Konzepte werden in diesem Kapitel separate Ebenen komponentenbasierter Systeme abgegrenzt und verschiedene Alternativen ihrer inhaltlichen Ausgestaltung gegenübergestellt und diskutiert. Ein Schwerpunkt liegt hierbei in der Untersuchung formaler Spezifikationen auf Basis logikbasierter Semantiken. Die detaillierte Betrachtung struktureller und verhaltensmäßiger Aspekte und die vorgenommene Einordnung einer Typisierung sollen dabei erste Grundlagen für ein Klassifikationskonzepts erarbeiten, wie es als zentraler Kern des hier vorgestellen generativen Konstruktionsansatzes in nachfolgenden Kapiteln dargestellt wird. Bevor jedoch zu solchen abstrahierenden Fragestellungen übergegangen werden kann, soll zunächst eine detaillierte Betrachtung der technisch motivierten Aspekte einer Komponentsicht erfolgen, um den Gegenstand der gewünschten Spezifikationen überhaupt erst einmal näher zu fassen.

2.1 Technisch motivierte Aspekte

Im Folgenden werden detailliertere konzeptionelle und formale Gesichtspunkte eines Baukastenprinzips sowie konkrete Mechanismen zur Umsetzung ablauffähiger Komponenten in realen Systemen betrachtet. Die Darstellung gliedert sich dabei zum

[1] Zum Modellbegriff: [Rumbaugh et al., 1991] definiert ein Modell etwa als »... an abstraction of something for the purpose of understanding it before building it.« Nach [Jackson, 1995] ist ein Modell »An abstraction of reality that [itself] shares important behaviors, properties, and structure with that reality.« [Gluch & Weinstock, 1998] unterscheidet entsprechend verschiedene Modell*arten*. Der Modellbegriff soll in dieser Arbeit nicht näher diskutiert werden, sondern im Sinne dieser Definitionen als abstrahierende, jedoch analoge Eigenschaften besitzende Sicht auf reale Systeme verwendet werden.

einen in die Komponenten selbst, sowie deren Interoperabiltät und ihrer Formalisierung.

Da nicht jeder technisch mögliche Baustein sinnvoll ist, stehen zunächst konzeptionelle Überlegungen bezüglich *strukturelller Aspekte* und verschiedener Ausprägungen der Komponenten selber im Vordergrund. Auch die *Anpassungsfähigkeit* eines Bausteins soll betont werden, da erst entsprechend flexibel konfigurierbare Einheiten die gewünschte *Wiederverwendbarkeit* sowie die Verringerung der Anzahl benötigter *Varianten* erlaubt.

Komponenten als Softwarebausteine sollen miteinander *interagieren*. Daher ist auch eine eingehende Untersuchung der Aspekte der *Integration* und *Interoperabilität* notwendig, die die Grundlage einer Softwarekonstruktion durch bloße *Kombination* bilden. Ein solches konstruktives Vorgehen bedarf dann jedoch einer Erörterung formaler Spezifikationsmöglichkeiten, mit denen der Abschnitt entsprechend endet.

2.1.1 Komponenten als Basisbausteine einer Softwareapplikation

Neben strukturellen Aspekten werden im Folgenden auch funktionale Anforderungen sowie die Frage der Granularität behandelt, um den notwendigen Charakter eines Anwendungsbausteins zu verdeutlichen.

2.1.1.1 Struktur, Art und Verhalten

Komponenten stellen zunächst einmal ein Strukturierungsmittel für Softwaresysteme dar. In diesem Zusammenhang können statische und dynamische Sichten unterschieden werden, d.h. insbesondere Strukturen zum Entwicklungszeitpunkt müssen nicht notwendigerweise den Strukturen zur Laufzeit einer komponentenbasierten Anwendung entsprechen. In jedem Fall sind die Komponenten selbst jedoch kapselnde, klar abgegrenzte Abstraktionen der inneren Applikationsstruktur mit formal eindeutig spezifizierten Schnittstellen als ausschließlichen expliziten Zugangspunkten (vgl. a. 1.1.6).

Abstraktion und Kapselung Komponenten sind in diesem Sinne für sich abgeschlossene (engl. *self contained*) Gebilde, die von einer gewissen Art einer Grenze (engl. *boundary*) umgeben sind. Dieser *Grenzbegriff* bildet die Grundlage der Strukturierung, der kontrollierten Interaktion und der Verifikation (vgl. a. 1.1.6) und

wird häufig auch als *Hülle* eines Bausteins bzw. als seine *Oberfläche* (engl. *surface*) bezeichnet.

Im Sinne einer Programmierung sollten Komponenten *Werte erster Klasse* (engl. *first-class-values*) darstellen, um geschlossene, homogene und aggregationsfähige Komponentenstrukturen zu erhalten. Diese Fragestellung wird später noch detailliert behandelt.

Schnittstellen Schnittstellen (engl. *interfaces*) stellen zunächst ein probates Hilfsmittel zur Materialisierung der obigen Abstraktionsgrenze dar. Sie stellen damit die Zugangspunkte zur Funktionalität eine Komponente dar und ermöglichen zugleich deren Untergliederung (zum Beispiel in Nutzer- bzw. Administrationszugriffe).

Wechselseitige Beziehungen innerhalb einer komponentenbasierten Applikation gründen sich dann auf solchen Schnittstellen und werden von den dahinterstehenden konkreten Komponenten entkoppelt, sodass im weiteren Verlauf dieser Arbeit häufig die Schnittstellen selbst in den Vordergrund der Betrachtung rücken.

Im hier vorgestellen Ansatz wird dabei dann von einer *Segmentierung*[2] gesprochen, die eine Aufteilung der gesamten funktionalen Oberfläche einer Komponente nicht nur als sinnvolle logische Untergliederung, sondern als Aggregation intern vollständig *unabhängiger* Funktionseinheiten betrachtet (vgl. a. Seite 60ff).

Es stellt sich die Frage, ob im Rahmen einer Komponentensicht ein vollständig neuer Schnittstellenbegriff benötigt wird oder ob die bestehenden Ansätze aus anderen Bereichen wie der Telekommunikation oder der Objektorientierung nicht bereits ausreichen. Beispielsweise unterscheidet das ODP-Referenzmodell (*Reference Model of Open Distributed Processing*) [Commission, 1995] zum Beispiel nach dem Grad der Abstraktion einer Schnittstelle drei allgemeine Formen:

- *Operationale Schnittstellen* – Operationen abstrahieren dabei eine bestimmte Funktion bzw. in ihrer Gesamtheit einen *Dienst* der zugehörigen Komponente. Operationale Schnittstellen bieten eine Menge[3] solcher Operationen an, die wie Prozeduren in imperativen Sprachen aufgerufen werden können.

[2] vgl. a. den Segmentbegriff in Kapitel 4

[3] oder mehrere (unter Umständen geordnete) Mengen

- *Stream-Schnittstellen* – Streams sind Folgen von Daten, die durch jeweils einen Namen und einen Typ beschrieben werden. Diese eignen sich besonders für Anwendungsbereiche mit sehr großem Datenaufkommen wie etwa dem Multimediabereich im Rahmen eines Erzeuger-Verbraucher-Modells (*engl. producer/consumer model*).

- *Signal-Schnittstellen* – Signale ermöglichen die elementarste Form der Interaktion mittels eines minimalen Informationsaustauschs zwischen einem initiierenden und einem reagierenden Objekt, was in der Regel in ereignisbasierten Systemen resultiert.

Diese drei Sichten beeinflussen auch maßgeblich den Entwurf des Klassifikationskonzepts dieser Arbeit. Dabei stellen insbesondere die *Datenflussorientierung* und die *Typisierung* der Stream-Schnittstellen, sowie die « inhaltslose», bloß benachrichtigende Kommunikation der Signal-Schnittstellen interessante Aspekte dar, die zu einer *Entkopplung* von Komponenten untereinander und die Konzentration auf deren *Interaktionen* führen. Kapitel 4 wird darauf zurück kommen.

Eine allgemeinere Unterscheidung kann auch folgendermaßen vorgenommen werden:

- *Grafische Schnittstellen* – Die Beschreibung erfolgt in grafischer Form (z.B. durch Diagramme) und kann allerdings häufig nicht automatisch verarbeitet werden.[4]

- *Programmiersprachenabhängige Schnittstellen* – Die Beschreibung der Schnittstelle erfolgt in der verwendeten Programmiersprache. Dies entspricht der pragmatischen Vorgehensweise wie sie viele Entwickler verinnerlicht haben und hat leider auch eine starke Prägung auf häufig allzu technisch orientierte Modellierungsnotationen wie etwa die UML genommen.

- *Programmiersprachenunabhängige Schnittstellen* – Derartige Schnittstellen werden durch unabhängige Beschreibungssprachen wie *IDLs* (*interface definition languages*) beschrieben. Dieses Verfahren hat sich in der Entwicklung verteilter Systeme durchgesetzt und sollte konsequent auch

[4] Diese zunächst vielleicht nicht ganz praktikabel erscheinende Variante einer Schnittstellenrepräsentation gewinnt zunehmend an Bedeutung, insbesondere im Trend einer *problemorientierten* Programmierung wie sie etwa das *Intentional Programming* von Microsoft vorsieht (s.a. Abschnitt 5.5.8).

Einzug in die lokale Entwicklung halten, da es einen wichtigen Schritt auf dem Weg zur *expliziten*, vom konkreten System abstrahierenden Darstellung erlaubt.

- *Selbstbeschreibende Schnittstellen* – Die Schnittstelle liegt dabei zunächst lediglich als abstrakte Einheit vor. Es sind dann Möglichkeiten vorhanden, die benötigten Informationen in einem interaktiven Prozeß entweder von der Schnittstelle selber oder ausgelagerten systemtechnischen Unterstützungsmechanismen (*engl. interface repository* dynamisch zu erfragen. Diese Sicht erscheint zwar auf den ersten Blick reizvoll, führt aber je nach Umsetzung wieder zu einer stärkeren *Implizierung* von Schnittstellenbeschreibungen (in den Programmcode hinein, vgl. beispielsweise die JavaBean-Introspektion [Vanhelsuwé, 1997]).

Letztere Sichtweise ist daher insbesondere dann sinnvoll einsetzbar, wenn sie über externe Mechanismen realisiert wird, wie es auch das in Kapitel 4 vorgestellte Typmanagment-Konzept vorsieht.

Inhaltlich sollten Schnittstellen eine – mehr oder weniger formale – Spezifikation der durch die Komponenten zur Verfügung gestellten bzw. benötigten Funktionalität und des bei deren Nutzung zu erwartenden Verhaltens darstellen. Zu diesem Zweck finden sich derzeit meist die aus der Objektorientierung bekannten Abstraktionen *Attribut*, *Ereignis* und *Methode* bzw. Mischformen mit wechselseitiger Emulation.[5] Diese typischen drei Kennzeichen der modernden Objektorientierung sagen dabei aber im Grunde nichts über die *Fachfunktion* einer Komponente aus, d.h., welche Problemstellung des Einsatzbereiches einer Komponente diese eigentlich löst. Diese technisch motivierten Kennzeichen sollten daher in einem entsprechenden Klassifikationsmodell für Komponenten abstrahiert werden (vgl. Kapitel 4), zumal sie in der Regeln keine Verhaltensbeschreibung beinhalten.

Für die Betrachtung von Schnittstellen ist schließlich noch das Auftreten von Einfach- bzw. Mehrfachschnittstellen wichtig, d.h., die Frage, ob eine einzelne Komponente mehr als eine Schnittstelle besitzen kann. Während Einfachschnittstellen leichter zu handhabende Strukturkonzepte erlauben, lassen Mehrfachschnittstellen intuitivere Modellierungsmöglichkeiten zu. Mehrfachschnittstellen implizieren jedoch in den gängigen Modellen der Objektorientierung innere Abhängigkeiten die die Komplexität im Falle einer vollständigen Spezifikation des frag-

[5] zum Beispiel »getter / setter«-Methoden für Attributmanipulationen

lichen Objekts (oder eben auch einer Komponente) deutlich erhöhen.

Daher werden Mehrfachschnittstellen in dieser Arbeit als reine Aggregation von (unabhängigen) Einzelschnittstellen betrachtet. Nach Meinung des Autors können nur durch eine solche Vermeidung impliziter Abhängigkeiten formale Konstruktions- und Verifikationskonzepte für ein Komponentenmodell greifen. Entsprechende Konzepte und die Thematik geeigneter Komponentenschnittstellen werden im weiteren Verlauf dieses Kapitels noch detailliert behandelt.

Statik oder Dynamik Die dynamischen Eigenschaften einer Komponentenkomposition sind ein wichtiger Aspekt, da diese eine Evolution sich im Einsatz befindlicher Softwaresysteme ermöglichen. Obwohl nicht Schwerpunkt der vorliegenden Arbeit sind während ihrer Erstellung auch Unterstützungmechanismen für die hierbei notwendige dynamische Komponentenaktualisierung innerhalb einer laufenden Applikation entwickelt worden [Martens, 1999, Benthien, 1999, Häming, 1999, Moehl, 1999b, Lapok, 2000].

In einem solchen Fall sind dann die Komponentenschnittstellen mehr als nur statische Verbindungspunkte, sondern bedürfen einer besonderen Beachtung als *aktive* Systembestandteile etwa in Form von *Konnektoren* wie sie in Kapitel 3 noch im Detail betrachtet werden.

Aggregation Die Fähigkeit durch Aggregation von Komponenten wiederum neue, im Rahmen des Modells gültige[6] Komponenten zu bilden, ist zwar intuiv Bestandteil einer Komponentensicht, jedoch nicht selbstverständlich realisierbar. Oft ist eine Aggregation im *Modell* möglich, während konkreten Umsetzungen in die Praxis durch die zur Verfügung stehenden Techniken Grenzen gesetzt werden. Zum Beispiel ist in der Regel ein Zusammenführen zweier Schnittstellen zu einer einzigen neuen im Rahmen einer Aggregation von Komponenten technisch nicht möglich.[7]

Auch für den Einsatz formaler Methoden ist das gewählte Aggregationsmodell von elementarer Bedeutung, zumal entspechende Verfahren in mehrstufigen Modellen besonders effektiv

[6] Ist diese Bedingung erfüllt, soll hier von einem *geschlossenen* Komponentenmodell gesprochen werden.

[7] Delegationsmechanismen seien hier unberücksichtigt, da sie eine zusätzliche Indirektionsebene einführen und somit kein homogenes Modell erlauben.

einsetzbar sind (vgl. a. Abschnitt 2.3.4). Ein Verzicht auf eine Aggregationsmöglichkeit würde einem Verzicht auf eine Anzahl von (Wieder-) Verwendungsmöglichkeiten gleichkommen, da »einstufige« Kompositionen, also solche, die Komponenten direkt zu fertigen Applikationen zusammenfügen, ohne dass diese wiederum als gültige Komponenten behandelt werden können, zu relativ speziellen Bausteinen mit dementsprechend engem Einsatzbereich führen.[8]

2.1.1.2 Funktionalität und Granularität

Eine wesentliche Problemstellung in der Komponentendiskussion ist die Frage nach der erforderlichen oder wünschenswerten *Granularität* einer Komponente - wann also ist ein Stück Software denn sinnvollerweise eine gültige Komponente? Diese Frage steht in engem Zusammenhang mit dem *funktionalen Umfang* einer Komponente.

In Bezug auf die An- bzw. Verwendung einer Komponente stellt ihre *Funktionalität* die wichtigste Eigenschaft dar. Komponenten erbringen eine gewisse Dienstleistung, auf Grund derer sie von Dienstnehmern mit entsprechenden Dienstanforderungen selektiert und eingesetzt werden. Ob die Funktionalität einer Komponente den Anforderungen des Verwenders genügt, ist bei komplexen Komponenten nicht immer leicht festzustellen. In Anlehnung an [Bott & Ratcliffe, 1992] sollen in diesem Zusammenhang drei wichtige Eigenschaften der Funktionalität unterschieden werden:

- *Anwendbarkeit* – Anwendbarkeit (engl. *applicability*) drückt die Wahrscheinlichkeit einer erneuten Verwendung relativ zur spezifischen Einsatzdomäne aus. Komponenten sollten generell eine hohe Anwendbarkeit besitzen. Sie ist jedoch in der Regel auf jeweils eine bestimmte Domäne (vgl. a. Abschnitt 5.3) beschränkt.

- *Generalität* – Generalität (engl. *generality*) meint die Breite möglicher Anwendungen einer Funktionalität, also deren *Generizität*. Obwohl eine erhöhte Generalität zu häufigerer Anwendbarkeit führt, würde das Bereitstellen zu vieler Funktionen einer Komponente den Lernaufwand für deren korrekten Einsatz sowie die von ihr beanspruchten Ressourcen unter Umständig aus ein unsinniges Maß erhöhen.

[8] Die generative Softwareerstellung dieser Arbeit sieht entsprechend ein mehrstufiges Modell vor (vgl. a. Kapitel 5).

- *Vollständigkeit* – Vollständigkeit (engl. *completeness*) bezieht sich auf die Erwartungen innerhalb einer bestimmten Anwendungsdomäne. Die Funktionalität einer Komponente ist dann vollständig, wenn sie alle Aspekte enthält, die in semantischem Sinne für ihren vorherbestimmten Anwendungskontext unabdingbar sind. Komponenten sollten also eine sinnvolle, semantisch abgeschlossene – im Sinne von in der fraglichen Domäne als eigenständiges Konzept identifizierte – Funktionalität erbringen (s.a. Abschnitt 5.3).

Der funktionale Umfang einer Komponente sollte weder zu klein noch zu groß sein. Abgesehen von der Forderung nach Vollständigkeit führen funktional zu kleine Komponenten dazu, dass zwar ihre Anwendbarkeit steigt, dass aber zugleich ihre Gesamzahl groß wird, was verworrene, komplizierte Anwendungssysteme zur Folge hätte

Wird über den Umfang von Funktionalität gesprochen, so tritt wie bei vielen qualitativen Größen das Problem der Quantifizierung auf. Geeignete allgemeine Metriken bzw. Maßstäbe sind dabei kaum zu finden, denn was ein (zu) »großer« oder »kleiner« Funktionalitätsumfang ist, hängt praktisch immer vom Einzelfall ab. Als brauchbares theoretisches Hilfmittel haben sich clusteranalytische Verfahren zur Identifikation von Komponenten und damit der Festlegung ihres Umfangs gezeigt [Düwel & Hesse, 2000]. Praxisbezogene, intutiv eingesetzte Granularitätmaße lauten häufig (vgl. [Griffel, 1998]):

- *feinkörnig*, passiv genutzt,
- *mittelkörnig*, meist (inter)aktiv und
- *grobkörnig*, übergreifend aktiv koordinierend

Um die gewünschte Untersuchung frei von der Problematik der Granularitätsfrage durchführen zu können und dennoch über ein exaktes Granularitätsmaß zu verfügen, orientiert sich der Granularitätsbegriff der vorliegenden Arbeit am *Interaktionspotential* eines Stück Software. Ein Softwarebaustein wird als Komponente im Sinne der vorgenommenen Begriffsbildung (vgl. Seite 20) bezeichnet, wenn er zu mindestens einer *Interaktion* (im Sinne des in Kapitel 4 entworfenen Klassifikationsmodells) fähig ist.

2.1.2 Integration und Interoperabilität

Die oben angesprochene Aggregation kann auch als Integration einzelner Bausteine zu komplexeren Systemen aufgefasst wer-

den, welche zunächst wieder Komponenten und auf oberster Ebene letztendlich dann fertige Anwendungen darstellen. Dabei ist nicht nur eine *Komposition* zu ermöglichen, die eine – möglicherweise benötigte – simultane Ausführbarkeit und die – möglicherweise entfernte – Kommunikation der beteiligten Komponenten unterstützt, sondern auch durch entsprechende *Interoperabilitätsmechanismen* ein gemeinsames semantisches Verständnis des resultierenden Informationsflusses zu sichern. Daneben besteht der Wunsch, den resultierenden koordinierenden Rahmen (das *Anwendungsskelett*) zu externalisieren, um die Anwendungs*struktur* reproduzierbar und nachvollziehbar vorliegen zu haben. Als Ausdrucksmittel kommen hier etwa *Skipt*- und *Koordinationssprachen* in Frage (vgl. [Griffel, 1998]).

2.1.2.1 Komposition

Grob kann zunächst zwischen *interner* und *externer* Komposition differenziert werden [Sametinger, 1997]. Dabei wird von interner Komposition gesprochen, wenn homogene Komponenten innerhalb eines einzelnen Softwaresystems etwa durch das Linken von Objektcode integriert werden. Bei der externen Form[9] liegen die Komponenten als unabhängig ausführbare Einheiten vor. Innerhalb dieser Grundkategorien finden sich wiederum zahlreiche unterschiedliche Ausprägungen. Im Folgenden werden kurz einige wichtige Formen der Komposition in Anlehnung an [Sametinger, 1997] und [Nierstrasz & Dami, 1995] beschrieben, da diese dem in Kapitel 5 vorgestellten generativen Ansatz als Kompositionsvariante gegenübergestellt werden können.

Textuelle Komposition Das textuelle Zusammenführen von Quellcode zur späteren Kompilierung kann als eine primitive Form der Komposition aufgefaßt werden. Ein gewisser grundlegender Grad der Abstraktion ist dann zum Beispiel durch *Makrotechniken* zu erreichen. Weitere Beispiele dieser Klasse sind *Templates* (zum Beispiel in C++) und *Generics* (zum Beispiel in ADA), bei denen eine Parametrisierung möglich ist und eine Verifikation während der »Komposition« durchgeführt wird.

Funktionale Komposition Ein funktionale Komposition ist in fast allen Programmiersprachen zu finden. Komponenten sind in diesem Fall parametrisierte, funktionale Abstraktionen, die

[9] Dies entspricht der in der vorliegenden Arbeit vertretenen Sichtweise.

durch Aufrufe, welche den Parametern entsprechende Argumente zuweisen, aktiviert bzw. instanziert werden. Da Funktionen selber in der Regel Werte erster Klasse sind – d.h. Funktionsparameter können selber wieder Funktionen sein – wird eine Komposition höherer Ordnung möglich, welche in gewissem Rahmen automatisiert werden kann. Der funktionale Ansatz basiert formal auf dem λ-Kalkül, und es existieren hochentwickelte Typsysteme mit mächtigen transparenten Inferenzmechanismen und weitreichenden Verifizierungsmöglichkeiten für Korrektheit von Kompositionen. Durch RPC-Mechanismen [Coulouris et al., 1994] wird eine systemübergreifende Komposition autonomer Teilkomponenten ermöglicht.

Objektorientierte Komposition Objektorientierte Komposition basiert auf den Prinzipien von Vererbung bzw. Delegation. Hierbei wird der Inhalt bestehender Komponenten übernommen und erweitert, wobei die Kompatibilität[10] gewahrt bleibt. Durch Polymorphismus und späte Bindung wird eine dynamische Form der Komposition ermöglicht.

Komposition durch Blackboards Blackboard Komposition, wie sie in klassischen verteilten Systemen zum Einsatz kommt, ist die indirekte Verbindung von Subsystemen über adressierbare, gemeinsam zugängliche Informationsablagen innerhalb eines globalen Raumes wie etwa gemeinsamer Speicher oder Kommunikationskanäle. Blackbord-Komposition erlaubt im Gegensatz zu *lokalen* Mechanismen wie der funktionalen Komposition die simultane Verbindung ganzer Komponentenmengen und stellt daher eine *globale* Methode dar. Den aus diesem Ansatz resultierenden sehr »losen« Kopplungen[11] stehen deutliche Einschränkungen bezüglich möglicher Korrektheitsprüfungen gegenüber.

In Theorie und Praxis anzutreffende weitere Konzepte wie Objektmodelle, Verbunddokumente, Verbundapplikationen, Integrierte Umgebungen oder Middleware-Plattformen stellen lediglich jeweils Spezialfälle der obigen Unterscheidungen dar, sodass auf sie hier nicht weiter eingegangen wird.

Unabhängig von konkreten Kompositionsmechanismen ist eine deutliche Differenzierung zwischen logischen und technischen Einheiten der Komposition vorzunehmen, da sonst technische

[10] Entsprechende Begriffdefinitionen erfolgen in Kapitel 4

[11] D.h. solchen, bei denen die gekoppelten Partner keine Kenntnis voneinander benötigen, sowie ihre Verbindung dynamisch auf- und abbauen können.

Gegebenheiten die konzeptionelle Ebene verwässern. Leroy unterscheidet in diesem Zusammenhang zwischen konzeptioneller *Modularisierung* als logischem Strukturierungsvorgang und *separater Kompilation* als technischem Übersetzungsmechanismus [Leroy, 1994].

Vor allem bei den internen Kompositionsformen besteht traditionell meist ein scharfer Schnitt zwischen separater Betrachtung von Komponenten vor der Komposition und monolitischen, optimierten Softwaresystemen danach.[12] Dies verhindert die in der Komponentensicht gewünschte Dynamik und Flexibilität, sodass dort Kompilierung und Bindung zunehmend nicht mehr als endgültige Verschmelzung einzelner Systembausteine aufgefaßt werden, sondern eben eine dingliche Bausteinsicht, in der die einzelnen Systemeinheiten auch in der Gesamtanwendung noch unterscheidbar und manipulierbar vorliegen.

2.1.2.2 Interoperabiltität

Teil des Kompositionsbegriffs ist neben dem Vorhandensein grundlegender Kommunikationsmechanismen ein gemeinsames Verständnisses bezüglich logischer Abläufe und semantischer Inhalte eines wechselseitigen Informationsaustausches im Zuge der (domänenspezifischen) Zusammenarbeit. Eine oft zitierte anschauliche Analogie basiert auf dem Elektrizitätsnetz verschiedener Länder. Zum einen passen die Stecker elektrischer Geräte oft nicht in Steckdosen fremder Länder, was durch entsprechende Adapter behoben werden kann. Zum anderen reicht diese Maßnahme, die als Komposition gesehen werden kann, aber trotzdem nicht aus, falls das fremde Elektrizitätsnetz eine andere Spannung liefert. In diesem Fall muss dann zusätzlich eine Transformation erfolgen, was entweder durch das Gerät selber oder ein separates Bauteil erfolgen kann.

Die Lösung von Interoperabilitätsproblemen ist bei Softwaresystemen auf Grund der Vielzahl möglicher Parameter eines solchen Systems eine wesentlich schwierigere Aufgabe als in vielen physikalischen Systemen der klassischen Industrie. Zum einen müssen die syntaktischen Unterschiede von Schnittstellen wie Namen von Methoden und Parametern überwunden werden. Zum anderen sind unterschiedliche semantische und funktionale Verhaltensweisen der beteiligten Komponenten anzugleichen. Dem obigen « Bauteil»entspräche in der Software dann ein *Konnektor*, der als *aktiver* Softwarebaustein eine entsprechend an-

[12] Die typische Situation einer objektorientierten Applikation.

gleichende Verbindung herstellt. Kapitel 3 zeigt die Ausprägung eines solchen Konzepts im Detail.

Allgemein wird die Fähigkeit von Softwarekomponenten, trotz unterschiedlicher Implementationssprachen, Schnittstellen und Plattformen bzw. Modellabstraktionen zu kommunizieren und zu kooperieren, als *Interoperabilität* bezeichnet [Konstantas, 1995].

Interoperabilitätsmechanismen für Komponenten setzen dabei an den Schnittstellen der einzelnen Bausteine an und unterteilen sich dabei je nach Strategie und Ansatzpunkt in verschiedene Kategorien. Kapitel 3 wird diese Aspekte nochmals aufgreifen, hier zunächst eine allgemeine Unterscheidungsmöglichkeit:

- Überbrückung von Schnittstellen – Die Strategie der *Schnittstellenüberbrückung* (engl. *interface bridging*) gleicht *angebotene* und *vorausgesetzte Schnittstellen* kooperationswilliger Partner durch Definition einer *Schnittstellentransformationssprache* (engl. *interface transformation language – ITL*) an. Die von der Programmiersprache abhängige ITL beschreibt detailliert, wie die Inhalte der beteiligten Schnittstellen wechselseitig ineinander überführt werden können.

- Standardisierung von Schnittstellen – Schnittstellenstandardisierung (engl. *interface standardization*) vereinheitlicht die allgemeinen Schnittstellen von Komponenten. Zu diesem Zweck finden sogenannte *Schnittstellenbeschreibungssprachen* (engl. *interface definition languages – IDL* Verwendung, welche unabhängig von den konkreten Programmiersprachen sind. Die abstrakten Schnittstellenbeschreibungen können mit Hilfe spezieller Compiler zu sprachspezifischen Schnittstellenrümpfen (engl. *interface stubs*) übersetzt werden, welche Implementationen der entsprechenden Schnittstellen darstellen und in konkrete Programme eingebunden werden.

Je nach Ursprung werden bei beiden Mechanismen in der Literatur noch die *prozedurorientierte* Interoperabilität, die am Punkt des Prozeduraufrufes ansetzt, und die *objektorientierte* Interoperabilität, die vom Punkt des Objektes, also mindestens einer ganzen Schnittstelle, ausgeht, unterschieden [Konstantas, 1995]. Letztere Sichtweise hat die ganzheitliche Sicht auf die Grenze einer Komponente im Modell dieser Arbeit motiviert – im Sinne des in Kapitel 4 eingeführten Interaktionspotentials.

2.1.2.3 Nebenläufigkeit

An dieser Stelle soll noch kurz auf die Problematik der Nebenläufigkeit eingegangen werden, da das zu entwerfende Typmodell (s. Kapitel 4) explizit auch von Synchronisationsaspekten einer Komponente abstrahieren soll. Zur detaillierteren Diskussion siehe auch [Griffel, 1998].

Zunächst steigert die Einführung von Nebenläufigkeit, die in der realen Welt dem Regelfall interagierender Systeme enspricht, potenziell erheblich die Modellierungsmächtigkeit einer Komponentenssicht. Zudem sind Komponentensysteme bei externer Komposition Konglomerate autonomer Einheiten, weshalb ihnen naturgemäß eine inhärente Nebenläufigkeit innewohnt.

Auf Grund der Orthogonalität von Nebenläufigkeit zu Paradigmen der Komposition und Wiederverwendung ist ihre Integration in ein Komponentenmodell nicht trivial [Papathomas, 1995]. Für eine Komponentensicht werden wiederum zwei Formen einer Nebenläufigkeit unterscheiden [Sametinger, 1997]:

- *Interne Nebenläufigkeit* (engl. *intraconcurrency*) meint die unabhängige Ausführung von Vorgängen innerhalb einer einzelnen Komponente.
- *Externe Nebenläufigkeit* (engl. *interconcurrency*) meint die unabhängige Ausführung verschiedener Komponenten.

Auf die interne Nebenläufigkeit wird häufig zur Vermeidung der damit verbundenen inhärenten Komplexität und Fehleranfälligkeit in der Komponentensicht verzichtet (vgl. a. [Griffel, 1998]). Die eingeschränkte Modellierungsmächtigkeit wird dann zu Gunsten einer besseren Praktikabilitität bewusst in Kauf genommen. Synchronisierende – in der Regel durch Serialisierung – Mechanismen werden dann durch die Laufzeit*umgebung* der Komponenten bereitgestellt, beispielsweise Applikations*container* (s.a. [Sun, 2000]).

2.1.2.4 Verteilung

Logische und physikalische *Verteilung* ist ein bedeutsamer Aspekt der Komponentensicht, da entsprechende Komponentensysteme oft als *verteilte Systeme* agieren sollen.

Coulouris et al. definiert ein entsprechendes verteiltes System als »*a collection of automous computers linked by a network, with software designed to produce an integrated computing*

facility« [Coulouris et al., 1994]. Vergleiche hierzu auch die Darstellung der *Koordinationsapplikationen* in [Griffel, 1998]. Historisch ist der Begriff der »Componentware« insbesondere auch aus der Client-Server-Diskussion der verteilten Systeme hervorgegangen [Stal, 1995].

2.1.2.5 Übergeordnete Strukturen und Rahmenwerke

Traditionelle Rahmenwerke (*engl. frameworks*) legen Anordnung und Beziehungen von Objekten untereinander fest und definieren auf diese Weise ein abstraktes Skelett – also die konzeptionelle Architektur – des angestrebten Gesamtsystems. Dabei liefert das Rahmenwerk jedoch in erster Linie Programmcode und entsprechende Architekturen werden lediglich *impliziert* – durch die Struktur des Rahmenwerks selbst, d.h., die Architektur liegt nicht explizit (sichtbar) vor, sondern verbirgt sich in der Codestruktur des Framework.

Im Sinne des Komponentenparadigmas erscheint der Begriff des Rahmenwerks aus einem etwas anderen Blickwinkel. Da die Einheiten des Frameworks – die Komponenten – schon in ihrer endgültigen, einsatzbereiten Form vorhanden sind und jeweils für sich bereits abgeschlossene Funktionalitäten darstellen, wird weniger ein abstrakter Rahmen aus nicht oder teilimplemetierten Klassen benötigt als mehr ein Regelwerk. Es sollte dann passender von einem *Kollaborationsrahmen* gesprochen werden [Griffel, 1998], der das Zusammenwirken dieser abstrakten Funktionalitäten repräsentiert und deren kompatiblen Austausch in einer konsistenten Weise ermöglicht.

Neben der Festlegung statischer Strukturen durch Rahmenwerke ist ebenfalls die dynamische Koordination globaler (also die eine gesamte aus Komponenten aufgebaute Anwendung betreffende) Abläufe zu regeln. Dies kann durch den Einsatz von *Koordinations*- bzw. *Skriptsprachen* erfolgen, die gelegentlich auch als *glue* – also amorpher Klebstoff – charakterisiert werden, der die beteiligten Komponenten zusammenhält [Nierstrasz & Lumpe, 1997]. Entsprechende Untersuchungen sind dann insbesondere für Generator-basierte Ansätze, wie sie hier etabliert werden sollen, von Bedeutung, da hier ein wesentlicher zu automatisierender Arbeitschritt das »Verkleben« einzelner Komponenten zu einer Gesamtapplikation ist. Die Kapitel 5 und 6 werden hierauf näher eingehen.

2.1.3 Formalisierung

Für die Etablierung von Automatismen wie sie für den hier angestrebten Ansatz einer generativen Softwarekonstruktion benötigt werden, ist die Formalisierung des zu Grunde gelegten Komponentenmodells unabdingbar, um zum einen mit systematischen Methoden deren korrekte Verwendung zu sichern und zum anderen deren Auffinden in geregelter Weise zu ermöglichen. Als vielversprechender Ansatz bietet sich hier die *Typisierung* von Komponenten an, der in Kapitel 4 nachgegangen wird. Zunächst sind jedoch Formalismen zu identifizieren, die überhaupt zu einer hierfür benötigten Spezifikation im Rahmen eines Komponentenmodells beitragen können.

Spezifikation vs. Typisierung

Da Softwarebausteine aus potenziell verschiedenen Entwicklungsprozessen stammen, wurden sie untereinander nicht notwendigerweise direkt auf eine wechselseitige Interaktion abgestimmt, dass heißt, es existiert kein gemeinsames Rahmenwerk. Es ist daher unumgänglich, die Regelgerechtheit wechselseitiger Komponentenbeziehungen während der Konstruktion einer Anwendung[13] zu prüfen und eine Abstimmung gegebenenfalls (indirekt) durchzuführen, was zu den Konzeptentwürfen der Adaption und der Klassifikation in den beiden folgenden Kapiteln führt. Diese Konzepte erlauben dann auch ein systematisches, automatisierte Retrieval geeigneter Bausteine aus einem potenziell großen Angebotsraum.[14]

Ein entsprechender Komponentenpool wird ebenfalls von dem in Kapitel 5 vorgestellten generativen Ansatz vorausgesetzt. Geeignete Komponenten können dabei entweder Alternativen oder Ergänzungen zu benötigten Bausteinen sein. Um entsprechende Vergleichskriterien nutzen zu können, ist einerseits eine möglichst exakte und damit in der Regel formale *Spezifikation* der Einzelbausteine notwendig, um deren Eignung für eine beabsichtigte Aufgabe entscheiden zu können, andererseits ein vergleichendes Konzept, das solche Spezifikationen klassifizieren kann, etwa durch die Einführung eines Typsystems, wie es Kapitel 4 vorstellt.

[13] und im Bedarfsfall auch bei einer dynamischen Rekonfiguration zur Laufzeit

[14] Sun Microsystems bewirbt beispielsweise ein *use & drop*-Konzept, das ein spontanes »Zusammensuchen« von Komponenten während der Applikationsentwicklung aus dem Internet vorsieht. Entsprechende Such- bzw. Klassifikationskriterien werden jedoch nicht vorgeschlagen.

2.2 Entwicklungsmethodik

Das Komponentenparadigma ist nur zum geringeren Teil eine technische Herausforderung. Für die Realisierung der impliziten Vorteile und Versprechungen einer Komponentensicht erfordern in erster Linie eine ganzheitliche abgestimmte Methodik der Softwareentwicklung. Dieser Gedanke komponentenorientierter Methodologie bildet die Grundlage für attraktive Visionen zukünftiger Softwareentwicklung.[15] So sieht zum Beispiel Nierstrasz für die Zukunft ein Szenario basierend auf »Softwareinformationssystemen« (SIS) voraus [Nierstrasz & Dami, 1995]. Diese SIS enthalten neben Beschreibungen konkreter Komponentenframeworks ein gebündeltes Anwendungswissen verschiedener Domänen und stellen in diesem Sinne Expertensysteme dar. Softwareentwicklung in idealisierter Form ist dann ein Prozeß, der auf informalem Dialog mit diesem Expertensystem basiert, in dessen Verlauf eine passende Domäne abgegrenzt wird, eine Bestimmung der Anforderungen basierend auf gespeicherten abstrakten Modellen und Richtlinien erfolgt, und schließlich konkrete Artefakte – d.h. vor allem Komponenten – mitsamt den notwendigen Anpassungen zu der gewünschten Applikation führen. Da von einem Wasserfallmodell abgesehen werden soll, beinhaltet das Szenario die Möglichkeit, einzelne Schritte später in erneuten Zyklen an veränderte Anforderungen anzupassen.

Das beschriebene Szenario ist noch zum großen Teil Fiktion und wirft zahlreiche Fragen bezüglich Umsetzbarkeit bzw. Praktikabilität auf.[16] Insbesondere ist der Autor hier der Ansicht, dass der Einsatz entprechend umfangreicher Expertensysteme nicht praktikabel und konsistent durchfürbar sein wird[17], sodass der Ansatz der vorliegenden Arbeit eine Dezentralisierung des Wissens präferiert, indem dieses als Teil der Komponentenbeschreibungen selbst sowie in Form expliziter Architekturbeschreibungen vorliegt.

---

[15] Auch der generative Ansatz dieser Arbeit versteht sich als Beitrag zukünftiger Entwicklungsformen von Softwaresystemen, indem er einen Gesamtprozess für die Anwendungsentwicklung angibt (vgl. Kapitel 5 und 6).

[16] Microsofts »Intentional Programming« [Simonyi, 1995] zeigt aber zum Beispiel bereits gangbare Wege in diese Richtung auch für die Entwicklungspraxis auf.

[17] Man denke etwa an das Vorhandensein zahlreicher Wissensbasen mit *unterschiedlichem* Wissensstand an verschiedenen Entwicklerstandorten.

Zu unterscheiden ist dann noch die Entwicklung wiederverwendbarer Komponenten von der Entwicklung von Anwendungen als Kompositionen vorgefertigter Komponenten. Um wiederverwendbare Strukturen zu identifizieren, tritt die Aufbereitung der jeweiligen *Anwendungsdomäne*, d.h. des fachlichen Problem- und Einsatzbereichs für den eine Applikation benötigt wird, in den Vordergrund. Abschnitt 5.3 geht daher im Rahmen der vorgeschlagenen Konstruktionsprozesses detaillierter auf die entsprechenden Fragestellungen des *Domain Engineerings* ein.

2.2.1 Komponentenentwicklung

Nachdem potenziell sinnvolle Komponenten im Rahmen der Domänenanalyse identifiziert wurden, können diese in konkret einsetzbaren Bausteine umgesetzt werden. Da diese dem Zweck späterer Wiederverwendung innerhalb einer Anwendungskomposition dienen, wird dann von einer »Entwicklung *für* eine Wiederverwendung« (engl. *development for reuse*) gesprochen. Die dabei auftretenden Fragestellungen sind jedoch nicht Gegenstand der vorliegenden Arbeit. Allgemeine Entwurfsrichtlinien für Komponenten sind zum Beispiel in [Mey, 1995] zu finden (vgl. a. [Griffel, 1998].

Bei Bausteinen, die umgekehrt aus einer Anwendungsentwicklung hervorgegangen sind, also nicht speziell als separate Komponenten entwickelt wurden, kann durch Generalisierung in Form von Erweiterung bzw. Einengung des Anwendungsbereiches, Separierung durch Isolation und verbesserte Konfigurierbarkeit die Verwendbarkeit nachträglich verbessert und über den ursprünglichen Einsatzkontext hinaus gehoben werden. Dieses Vorgehen entspricht dem derzeit in der Entwicklungspraxis noch sehr häufigen Vorgehen, mit dem sich entsprechende Entwicklungsteams an eine komponentenbasierte Denk- und Handlungsweise herantasten.

Allgemein sind erfolgreiche Komponenten das Ergebnis eines längeren Reifeprozesses über mehrfache Anwendungszyklen.

2.2.2 Anwendungsentwicklung

Die komponentenbasierte Entwicklung von Anwendungen basiert primär auf der Komposition vorgefertigter Bausteine und stellt somit ein »Entwickeln *mit* Wiederverwendung« (engl. *developing with reuse*) dar. Wichtig ist, dass die Entwicklung dabei von Anfang an mit der expliziten Intention des Einsatzes bereitstehender Komponenten erfolgt. So sollte die Suche nach ge-

eigneten Komponenten nicht erst nach der Analyse der Anwendungsproblematik einsetzen, sondern bereits aktiver Teil dieser Phase sein, um die Zahl notwendiger *zusätzlicher* Implementationen gering zu halten und eine Orientierung eines Systementwurfs an bewährten vorhandenen Bausteinen zu ermöglichen. Die dabei auftretenden Problematik der Vorwegnahme von Entwurfsentscheidungen durch die Eigenschaften vorhandener Komponenten mit der Gefahr »am Anwendungsproblem vorbei zu entwicklen« und eine wenig problembezogene Lösung zu erhalten, kann umgangen werden, wenn einzelne Bausteine direkt mit einzelnen Fachkonzepten der fraglichen Domäne gleichgesetzt werden. Dann entsprechen die Ergebnisse der Aufbereitung einer Domäne als sinnvolle Entwurfsentscheidungen des jeweiligen Anwendungsbereichs auch genau den verfügbaren Komponenten. Der generative Konstruktionsprozess in Kapitel 5 betont daher ein solches Vorgehen.

Ein intuitiver Bestandteil des Komponentenparadigmas ist die *Evolution* von Softwaresystemen, da ein Austausch oder eine Veränderung der lokal gut abgegrenzbaren Einzelbausteine mit ihrer losen Kopplung naheliegt. Daher sind entsprechende Entwicklungsprozesse auf jeden Fall zyklisch und beziehen fertige bzw. teilweise fertig gestellte Systeme im Sinne einer partizipativen oder explorativen Entwicklung[18] deutlich stärker ein als herkömmliche Vorgehensweisen der Softwareentwicklung. Im Rahmen solcher Entwicklungsmodelle erlaubt der Einsatz des in Kapitel 5 vorgestellten Generatorkonzepts insbesondere auch schnelle Komponenten*tests*, indem diese in einer Gesamtanwendung ihre Funktionalität und Interaktionsfähigkeit unter Beweis stellen müssen, ohne dass ein Entwickler umfangreichen Testcode schreiben muss, der in der Regel nicht für die eigentliche Anwendungserstellung genutzt werden kann.

Festzuhalten bleibt die enge Verzahnung der drei Aspekte Domain-Engineering, Komponenten- und Anwendungsentwicklung in der Komponentsicht im Allgemeinen und dem Ansatz dieser Arbeit im Besonderen.

[18] Also der frühzeitigen Einbeziehung des Endkunden in die Anwendungsentwicklung.

2.3 Struktur- und Modellsicht des Komponentenbegriffs

Die weitere Darstellung beschäftigt sich mit einer detaillierten Untersuchung der Aspekte und Eigenschaften einer abstrakten Sichtweise auf den Komponentenbegriff und der aus ihr resulierenden Systemstrukturen – die dann als Architekturmodelle bezeichnet werden – zur Vorbereitung eines Komponentenmodells, wie es dieser Arbeit zu Grunde gelegt werden soll. Der Schwerpunkt dieses Abschnitts liegt dabei auf Fragestellungen der Spezifikation (vgl. Abschnitt 2.1.3).

2.3.1 Vorüberlegungen

Eine erste abstrakte *Modellebene* betrachtet aus übergeordnetem Blickwinkel zunächst die eigentlichen Wesensmerkmale von Komponentensystemen, ohne jedoch auf deren Umsetzung einzugehen. In dieser Sicht werden informale Begriffe, Vorstellungen und Konzepte sowie deren Beziehungen diskutiert.

Die zweite Betrachtungsebene enthält Umsetzungen der erarbeiteten abstrakten Modellbegriffe in konkretere Beschreibungsformen und wird daher hier als *Beschreibungsebene* bezeichnet. Gegenstand der Diskussion sind dort Formalismen und Notationen, die geeignet erscheinen, eine Spezifikation der identifizierten Modellbestandteile zu unterstützen. Die Konstrukte dieser Ebene sind immer noch von relativ informaler Natur und erlauben daher auch noch keine direkte, automatische Verarbeitung. Diese wird erst von der letzten Betrachtungsebene ermöglicht, die eine konkrete Umsetzung in rechnerspezifische Ansätze, d.h. konkrete, ausführbare und manipulierbare Repräsentationen umfasst. Sie wird dementsprechend als *Repräsentationsebene* bezeichnet und erst beim endgültigen Entwurf eines Komponentenmodells in Abschnitt 2.3.4 wieder aufgegriffen.

2.3.1.1 Ausprägungen eines Komponentenmodells

Die Aspekte des Komponentenbegriffs beinhalten zunächst eine Reihe konzeptioneller Modelle. Innerhalb dieser Modelle werden insbesondere die konstituierenden Konzepte einer Komponente wie Schnittstellen, Attribute oder Nachrichten etc. erfasst. Ein *Strukturmodell* beschreibt in diesem Zusammenhang die statischen Eigenschaften der wechselseitigen Beziehungen zwischen diesen Einzelkonzepten. Ein *Verhaltensmodell* geht auf deren dynamische Interaktion und Semantiken ein. Daneben tritt noch

ein *Metamodell*, das eine abstrahierende übergeordnete Sichten erlaubt, zum Beispiel im Vergleich zu anderen Komponentenmodellen. Für eine praktische Handhabung werden dann *Spezifikationen* der einzelnen Modellabstraktionen benötigt. Wesentlich ist hier neben klassisch *statischen* Formalisierungen von Komponenten- und Kompositionsbegriffen die möglichst weitgehende Einbringung *dynamischer Aspekte* von Konglomeraten (Kompositionen) und der *Semantik* einer Bausteinfunktionalität.

Bei der Bestimmung geeigneter Formalismen ist vor allem von Bedeutung, dass die konzeptionellen Inhalte der zunächst informalen Komponentenmodelle möglichst unverfälscht und vollständig abgebildet werden. Dass dies bei einer Formalisierung – zumindest nach heutigem Wissensstand – nicht umfassend möglich ist, versteht sich von selbst. Die Formalisierung wird daher immer nur Teilbereiche umfassen können und selbst dort bis auf wenige Ausnahmen vereinfachende Annahmen und Idealisierungen beinhalten, da der Schwerpunkt der vorliegenden Arbeit auf deren Einsatzfähigkeit in der Entwicklungspraxis liegt.

2.3.1.2 Anforderungen

Es stellt sich die Frage, welche Anforderungen an abstrakte Systemstrukturen allgemein wünschenswert sind. Im Folgenden werden potenziell sinnvolle Eigenschaften diskutiert.

- **Multidimensionale (De-)Komposition** – *Komponenten sollen die Zerlegung von Softwaresystemen erlauben und dabei möglichst in multidimensionaler Weise orthogonale Aspekte berücksichtigen.*

Die Komponenten eines Anwendungssystems (Applikation) sollen dabei weitgehend unabhängig zu diskutierende aber eben sich ergänzende Aspekte [Aksit & Tekinerdogan, 1998] einer Anwendung repräsentieren und zwar sowohl in einer fachlichen Tiefe als auch in einer (Einsatz-) Breite. Dies entspricht der Forderung nach einer »Trennung der Belange« (engl. *separation of concerns*) [Czarnecki & Eisenecker, 2000], deren Diskussion auch Gegenstand des Kapitel 5 ist.

Komponenten sollen dabei zunächst grundsätzlich dem Leitbild einer *Bausteinmetapher* genügen. Wie die Bestandteile eines Baukastensystems – etwa der berühmten LEGO-Steine – sollen sie konstitutive abgeschlossene Einzelteile darstellen und sich im Rahmen bestimmter wohldefinierter Regeln zu einem Gesamtsystem zusammenfügen lassen. Diese oft zitierte Art der Bausteinmetapher ist hier allerdings nicht wirklich sinnvoll, da

LEGO-Bausteine nur *strukturelle* Schnittstellen festlegen, nicht aber verhaltensorientierte. Und anstelle »wohldefinierter Regeln« liegt eher kreative Grenzenlosigkeit für deren Einsatz vor. Letzteres liegt dabei in der Minimalität der gebotenen »Fachfunktion« – der Stecknorm seiner Oberflächengestaltung – eines einzelnen Bausteins begründet.

- **Geschlossenheit –** *Die Komposition von Teilkomponenten soll wieder in einer vollwertigen Komponente resultieren.*

Komponentenmodelle sollten bevorzugt *mehrstufiger* Natur sein. Die Unterscheidung von Komponentenaggregartionen und Komponenten ist dann nur eine Frage der Sichtweise. Insbesondere ist dann auch das fertige Softwaresystem (die Applikation) eine weiter- bzw. wiederverwendbare Komponente, was deren evolutionären Aus- und Umbau konzeptionell erleichtert. Geschlossene Modelle erlauben zudem einen homogeneren Einsatz formaler Methoden zu ihrer Spezifikation, da die Anzahl der zu unterscheidenden Systembestandteile verringert wird.

- **Kontextunabhängigkeit –** *Komponenten sollen frei von Seiteneffekten auf andere Komponenten des gleichen Aggregats sein.*

Es dürfen keine versteckten Abhängigkeiten, um eine generelle (Wieder-) vgl. Abb. 4.3erwendbarkeit und die explizite Erfassbarkeit aller Systemeigenschaften gewährleisten zu können. Dieser Punkt wird in der nächsten Forderung noch weiter konkretisiert.

- **Kapselung –** *Komponenten sollen für sich geschlossene Abstraktionen darstellen und sich durch eine klare Grenze von ihrer Umgebung absetzen.*

Die essentiellen Eigenschaften sowohl der selbst erbrachten, als auch der vom Kontext vorausgesetzten Funktionalität einer Komponente, sollen von außen über eindeutige, klar definierte Schnittstellen spezifiziert sein. Interne Details sind möglichst weitgehend zu kapseln, wobei sich der Striktheitsgrad an dem Gesamtmodell orientieren muss.

Die sinnvolle Striktheit einer Komponentenkapselung ist eng mit der Aussagekraft der verwendeten Schnittstellenspezifikation verbunden. Während allgemein schwach gekapselte Whitebox-Ansätze meist grundsätzlich verworfen werden, ist die vollständige Abgrenzung bei Blackbox-Ansätzen umstritten, da im Falle nicht spezifizierbarer interner Kontextabhängigkeiten

unkalkulierbare Seiteneffekte eintreten können. Werden letztere jedoch wie im Ansatz dieser Arbeit kategorisch ausgeschlossen, resultiert ein effektiv handhabbares Komponentenmodell. Die Thematik wird später noch eingehender behandelt.

- **Anpaßbarkeit bzw. Adaptability –** *Komponenten sollen in einem vom Hersteller vorgegebenen Rahmen variabel und anpassbar sein.*

Die Eigenschaft der Anpassungsfähigkeit fördert die Wiederverwendbarkeit von Komponenten in erheblichem Maße. Gerade weil Komponenten innerhalb eines domänenspezifischen Baukastensystems meist im Vorfeld der eigentlichen Anwendungserstellung entstehen, können nicht alle exakten Umstände des späteren Einsatzes vorausgesehen werden, und selbst die absehbaren Szenarien würden zu einer quantitativ unsinnig großen Menge individueller Varianten mit lediglich marginalen Unterschieden führen.

- **Zustand und Identität –** *Komponenten sollen langlebig sein und sowohl Identität als auch Zustand besitzen.*

Die Forderung nach individueller veränderlicher Beschaffenheit führt dazu, dass Komponenten keinen im mathematischen Sinne funktionalen Charakter besitzen, sondern ihr Verhalten im Laufe der Zeit und abhängig von ihrem Zustand verändern können. Komponenten sind dann keine abstrakten, statischen Schablonen, die beliebig instanziert oder dupliziert werden können und in einheitlicher Weise einen immer gleichen Dienst erbringen.[19] Sie sind vielmehr dynamische, persistierbare Laufzeiteinheiten, die flexibel anpassbar sind und mit konkreten, realen Konzepten der jeweiligen Anwendungsdomäne korrespondieren.

Daher spricht der Autor in diesem Zusammenhang auch von *dinglichen* Komponenten, d.h., Komponenten haben eine persistente Existenz unabhängig vom umgebenden Kontext, wodurch die Möglichkeit der individuellen Manipulation und Speicherung besteht.[20] Sie erzeugen dann zur Laufzeit nicht etwa Instanzen im Sinne der Objektorientierung, sondern werden als Individuen *aktiviert*.

- **Offenheit –** *Komponentenmodelle sollen die Integration heterogener Komponenten unterstützen.*

[19] vgl. daher auch die Beachtung einer Wertsemantik bei einem Komponentenvergleich über das Typmodell in Kapitel 4

[20] Diese elementare Eigenschaft wird nicht in jeder Komponentensicht geteilt, vgl. etwa [Szyperski, 1998].

Die Möglichkeit, heterogene Komponenten, d.h. solche, die sich unterschiedlicher Modelle und / oder Implementationstechniken bedienen, gemeinsam zur Applikationserstellung verwenden zu können, erlaubt die Auswahl aus einem maximalen Pool individueller Varianten und trägt den realistischen Annahmen der Entwicklungspraxis Rechnung, die im Laufe der Zeit stets wieder neue Ansätze und Techniken hervorbringen wird. Unrealistische globale Absprachen über notwendige Standards können so auf ein praktikables Maß reduziert werden. Weiterhin bildet diese Forderung eine Voraussetzung, um informationstechnische Altlastsysteme innerhalb neuer Technologien integrieren zu können, was Umstellungsprozesse erheblich erleichtert bzw. überhaupt erst möglich werden lässt.

Um den (komlexen) Fragestellungen einer Integration gerecht zu werden, müssen Bausteinspezifikationen über die in der bisherigen Programmiermethodik üblichen statischen Strukturinformationen hinausgehen. Es sollten in möglichst weitgehender Weise insbesondere auch *Verhaltensmuster* eingebracht werden. Ohne entsprechende Informationen ist eine sichere, womöglich automatisierte Verwendung von gekapselten Code-Bausteinen nicht möglich, da deren Anwendungsweise nicht alleine aus ihren strukturellen Eigenschaften hervorgeht. Ferner sollte es auch möglich sein, qualitative Eigenschaften von Komponenten wie Zuverlässigkeit/Robustheit oder Performanz/Ressourcenbedarf und Vielseitigkeit/Wiederverwendungswahrscheinlichkeit einzubringen, um deren Eignung besser – und automatisierbar – abschätzen zu können. Die Beahndlung dieser Kriterien liegt jedoch ausserhalb des Rahmens der vorliegenden Arbeit (siehe jedoch auch Abschnitt 5.9).

- **Nebenläufigkeit –** *Komponenten sollen nebenläufig und potenziell verteilt agieren.*

Da Komponentensysteme naturgemäß Konglomerate (Kompositionen) autonomer Subsysteme darstellen, liegt es nahe, deren unabhängige Aktivität in Zeit und Raum als Grundlage zu fordern. Dies wird auch durch die enge Verzahnung von konkreten Ausprägungen innerhalb der Bereiche Komponenten- und Parallelarchitekturen sowie Verteilung, Vernetzung, Internet etc. untermauert. Dies wiederum legt dann die Unterstützung einer asynchronen Entkopplung nahe (vgl. a. das Typmodell in Kapitel 4).

Die Vielzahl alleine der hier angeführten Anforderungen mit der unterschiedlichen Komplexität ihrer Fragestellungen lässt

die Verwendung einheitlicher Mechanismen und Konzepte für alle Aspekte unrealistisch erscheinen. Speziell im Bereich der Spezifikation steht den zur vollständigen Erfassung eine Komponente notwendigen komplexen Beschreibungen der Wunsch nach einer abstrahierenden, übergeordneten Typisierung entgegen. So eine allgemeine Klassifikation erlaubt den praktischeren Umgang mit Komponenten beispielsweise bei deren Suche zur Entwicklungszeit und behindert nicht durch Überspezifikation. Zur Lösung dieser Problematik bietet sich die Verwendung einer oder mehrerer expliziter Metaebenen mit jeweils für verschiedene Zwecke optimierten Konzepten an. Diese Einsicht resultiert in der parallelen Verfolgung beider Ansätze und der Einführung eines *mehrstufigen* Typmodells in Kapitel 4.

2.3.1.3 Methodik

Die Suche nach einer komponentenorientierten Methodik umfasst die Frage, auf welche Weise Komponenten zur Konstruktion von Softwaresystemen bzw. zur Dekomposition von Problembereichen eingesetzt bzw. angewandt werden sollen.

Diesbezüglich müssen zumindest zwei Varianten unterschieden werden. Zunächst könnte eine an den klassischen Methoden der Objektorientierung angelehnte Sicht eingenommen werden, bei der Komponenten – wie die Klassen von Frameworks – als Ausgangspunkt für schrittweise Spezialisierungen mit »Vererbung« ihrer Funktionalität dienen. Diese schrittweise Spezialisierung wird auch als Verfeinerung (engl. *refinement*) bezeichnet, beispielsweise der Konkretisierung eines Konzeptes wie im Beispiel: `Fahrzeug` → `KFZ` → `PKW` → `Stufenhecklimousine`.

Eine zweite, dinglichere und eben dieser Arbeit zu Grunde liegende Sicht könnte hingegen stärker die Identität und »Einzigartigkeit« individueller Bausteine betonen, wobei Komponenten dann in konzeptuell unveränderter Form aggregiert werden und konstituierende Teile eines aus ihrem Kompositum hervorgehenden Ganzen darstellen, zum Beispiel `KFZ = ( Karosserie + Motor + Rad1 ...)`.

Nach heutigem allgemeinem Verständnis ist letztere Sichtweise weit verbreitet. Der Verfeinerungsgedanke tritt im Zusammenhang mit Komponenten eher selten auf. Ein Grund mag darin liegen, dass der damit eng einhergehende Vererbungsbegriff negative Assoziationen mit den Schwachstellen objektorientierter Methoden weckt, welche doch – so die verbreitete Hoffnung – durch das Komponentenparadigma gerade umgangen werden sollen. Ob dies begründete Argumente sind – ob also ein Refine-

ment lediglich im Rahmen eines Whitebox-Reuse sinnvoll ist – muss eingehender untersucht werden. Sollten verträgliche Methoden der Verfeinerung existieren, würde deren Integration in die Methodik des Komponenteneinsatzes, wie sich zeigen wird, eine sinnvolle Ergänzung darstellen.

Es soll nun in einer praktisch orientierten Sichtweise der konkrete Entwicklungsprozess von Software unter Einfluß des Komponentenparadigmas betrachtet werden. Es sind dabei zwei konträre Vorgehensweisen zu unterscheiden. Zunächst wäre es denkbar, den Komponentengedanken als rein operationales Prinzip zu verstehen und in diesem Sinne bei jedem neuen Softwareprojekt eine allgemeine Top-Down Dekomposition mit individueller Ableitung spezifischer Komponenten durchzuführen. Eine strategische Sicht muss demgegenüber komponentenorientierte Softwareentwicklung als langfristigen, übergreifenden, lediglich im Hinblick auf Domänen spezifischen Prozess betrachten. Dieser startet dann mit einem initialen Domain-Engineering, das die Komponenten bestimmt und die spätere Wiederverwendung erlaubt. Der wesentliche Unterschied dieser beiden Sichten, welcher sich entscheidend auf die Konzeption eines geeigneten Komponentenmodells auswirkt, liegt im kausalen Zusammenhang von Komponentenerstellung und -verwendung. Die Vorstellung eines vorhandenen »Komponentenbaukastens«, aus dem vorgefertigte Komponenten zur Konstruktion eines neuen Systems entnommen werden, stellt gänzlich andere Voraussetzungen – zum Beispiel an die Komponentenspezifikation – als ein Ansatz, bei dem System- und Komponentenentwurf simultan einhergehen.

Da der heutige Componentware-Begriff von vorgefertigten Bausteinen ausgeht, soll dies die im weiteren Verlauf bevorzugt eingenommene Sicht sein. Die bestehenden Artefakte solcher Ansätze müssen entweder wie schon angesprochen variabel und anpassbar sein, oder noch gar nicht als *konkrete Komponenten* im eigentlichen Sinne auftreten, sondern zunächst nur als deren *Beschreibung* oder *Typangabe*. Hier könnte dann das oben angedeutete Refinement dazu beitragen die Grenzen allgemeiner Anwendbarkeit auszuweiten.

2.3.1.4 Integration einer Refinement-Sicht

Wird die methodologische Integration einer Bausteinverfeinerung akzeptiert, könnte Refinement durch Umgehung der damit im klassischen Sinne immer einhergehenden Beziehung zu Ver-

erbungskonzepten potenziell in eine mit dem Komponentengedanken harmonierende Form gebracht werden.

Ein mit dem Komponentansatz harmonierendes Refinement-Konzept könnte dadurch realisiert werden, dass nicht eine Komponente komplett übernommen und anschließend verändert wird – wie dies im Sinne einer klassischen Vererbung der Fall wäre –, sondern im Gegensatz dazu diese Komponente unter Beibehaltung ihrer Importbeziehungen komplett ausgetauscht wird.[21] Ein Refinement-Prozess findet dann unmittelbar am Punkt der Implementierung statt, die die zu verfeinernde Funktionalität (die exportierte Funktionalität des gesamten Aggregats) erbringt. Diese Vorgehensweise verletzt die Komponentensicht im Gegensatz zu einer Vererbung mit Overwriting nicht, da sie einem *Reassembling* der Einzelkomponenten entspricht.

Bei der Entscheidung, ein Refinement in ein Komponentenmodell zu integrieren, sind neben sauberer methodologischer Einbettung noch die resultierenden Konsequenzen für Komponentenschnittstellen und -spezifikationen im jeweiligen Modell zu beachten – darauf wird im Verlauf dieses Kapitels noch detaillierter eingegangen.

2.3.2 Die Modellebene

2.3.2.1 Strukturmodelle

Der folgende Abschnitt betrachtet die Aspekte komponentenspezifischer Strukturmodelle. Hierzu zählen vor allem die statischen Komponentenbestandteile, deren Gesamtheit zu einem allgemeinen Komponentenbegriff führt, sowie die wechselseitigen Beziehungen dieser Teile aber auch der resultierenden Komponenten untereinander.

Eine Basisrelation für Strukturen Die Struktur eines gewählten Komponentenmodells spiegelt insbesondere auch die Beziehungen der an einer Komposition beteiligten Komponenten wider. Diese Beziehungen der Bausteine untereinander können informell etwa als »verwendet«, »setzt sich zusammen aus« oder »ist aggreggiert« beschrieben werden. Mathematisch stellen solche Beziehungen Relationen dar. Bei der Abbildung struktureller Komponentenbeziehungen auf Relationen ergeben sich für letztere einige Bedingungen, die dem Autor für eine adäquate Modellierung notwendig erscheinen.

[21] Zu den Begriffen des Im- und Exports siehe Abschnitt 2.3.4.

Definition 2.1 (Basisstrukturrelation) *Eine grundlegende, mathematische Relation für Komponentenstrukturen sollte folgende Eigenschaften besitzen:*

- *Irreflexivität*
- *Antisymmetrie*
- *Transitivität*

Zunächst sollte *Irreflexivität* gefordert werden, da Komponenten sich nicht selber rekursiv enthalten können.[22] Diese Eigenschaft ist im geforderten dinglichen Charakter der Komponenten begründet.

Ferner ist die *Antisymmetrie* der Relation zu fordern, da Komponenten sich nicht wechselseitig auseinander zusammensetzen können. Dies gilt allerdings nicht für die informale Sicht der gegenseitigen *Nutzung*. In letzterem Sinne bedeutet es nur, daß solche Nutzbeziehungen stets von einer Einheit kontrolliert und koordiniert werden. Die Thematik wird später noch eingehender untersucht.

Die Eigenschaft der *Transitivität* weist darauf hin, dass eine Komponente a, welche eine weitere Komponente b als direkte Teilkomponente enthält, indirekt auch eine Komponente c als Bestandteil beinhaltet, aus welcher b komponiert ist. Die Transitivitätseigenschaft entfällt bei einer atomaren Modellsicht (vgl 2.3.4), wie sie gewählt werden kann, um die *Abstraktionsbarriere*[23] zwischen den Systembestandteilen zu betonen.

Da die angegebene Basisrelation antisymmetrisch ist, sind die zugehörigen Graphen gerichtet. Werden diese *Komponentenstrukturgraphen* betrachtet, so ist für den Begriff einer *Applikation* noch folgende Einschränkung vorzunehmen: In einer Applikation (also komponentenbasierten Struktur) existiert nur genau eine *Wurzelkomponente* (vgl. a. Abschnitt 6.4.3 bzw. Kapitel 5). D.h. eine Komponente, die nicht konstituierender Teil eines anderen Bausteins ist, begründet eine Applikation (s. Abbildung 2.1.a). Die Begriffe »Wurzelkomponente«, »Toplevel-Komponente« und »Applikation« bzw. »Anwendung« werden im

[22] Funktional ja, aber eben nicht strukturell, wenn eine entsprechende individuelle *Identität* jeder einzelnen Komponente vorausgesetzt wird (»Ist ein Baustein eingesetzt worden, steht er nicht mehr zur Verfügung.«).

[23] Der Begriff soll den Übergang im Abstraktionsgrad verdeutlichen, der auftritt, wenn nicht mehr ausschließlich auf der Ebene von Komponenten diskutiert wird, sondern deren konkrete Schnittstellen mit Methoden etc. betrachtet werden.

weiteren Verlauf daher als diesbezügliche Äquivalente verwendet.

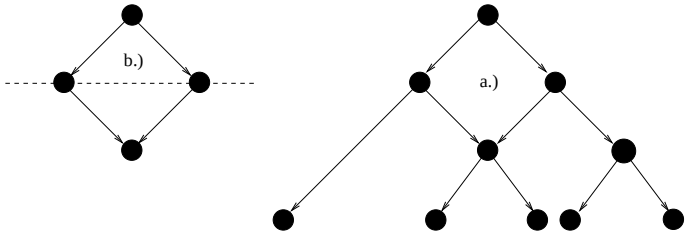


Abbildung 2.1 *Grundlegende Komponentenstrukturen*

Strukturelle Beschränkungen Die bisher beschriebenen Strukturmerkmale ergeben einen sehr breiten Rahmen möglicher konkreter Modelle. Es stellt sich die Frage, ob in Bezug auf komponentengerechte Modellstrukturen möglicherweise Einschränkungen notwendig sind. Insbesondere scheint die mehrfache Verwendung derselben bzw. gleichen Komponenten sowie das Auftreten zyklischer Strukturen einer Regelung zu bedürfen.

Mehrfachverwendung von Komponenten Von der ideellen Vorstellung her legt die Benutzung individueller Komponenten als Teile von Anwendungen deren *Verbrauch* nahe (»dingliche« Sicht). Eine solche Komponente könnte bei dieser Vorstellung nicht erneut zur Komposition eingesetzt werden. Tatsächlich ist die mehrfache Verwendung aber notwendig, denn eine als Baustein realisierte, exklusive Ressource – etwa eine Datenbasis – darf nicht durch ihre Nutzung innerhalb einer Anwendung für andere Zwecke unzugänglich werden.

So eine Mehrfachverwendung könnte dabei entweder innerhalb einer (s. Abbildung 2.1.b) oder zwischen verschiedenen (s. Abbildung 2.2) Komponentenstrukturen bzw. Applikationen stattfinden. Eine applikationsübergreifende Nutzung ein und derselben Komponente wird im Ansatz dieser Arbeit ausgeschlossen. Der oben angesprochenen Problematik einer gemeinsamen Ressourcennutzung (etwa einer Datenbank) wird dabei durch die Einführung jeweils individueller *Zugriffskomponenten*[24] als Abstraktion für alle derartigen Ressoucen begegnet.

Die multiple Verwendung einer einzelnen individuellen Komponente setzt deren Fähigkeit zur Handhabung nebenläufiger

[24] auch als Wrapper oder Ressourcenkomponenten bezeichnet

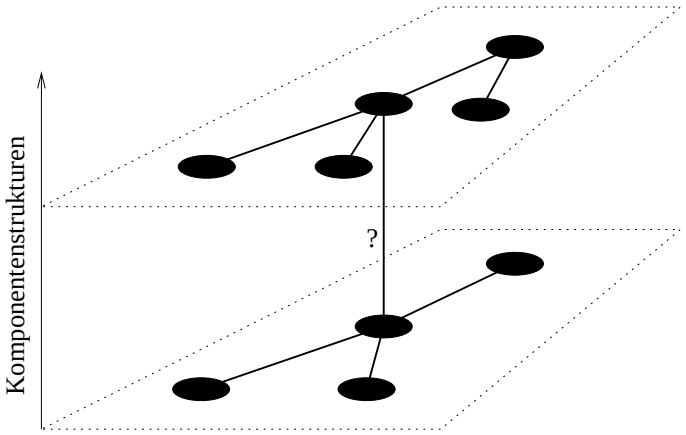


Abbildung 2.2
Applikationsübergreifende Komponentennutzung

Dienstzugriffe voraus und sollte somit stets ein Sonderfall bleiben.

Die hier vorgestellte Komponentensicht geht von individuellen Instanzen aus. Wird dann von einem Komponenten*typ* gesprochen, darf jedoch nicht die Analogie zum Klasse-Instanz-Verhältnis der Objektorientierung gezogen werden. Dort würde die Definition einer Klasse beispielsweise den Zugriff auf eine Datenbank beinhalten. Auf diese Datenbank würden dann alle konkreten Objekte (Instanzen der Klasse) gemeinsam zugreifen. Eine solche implizite Propagation von Abhängigkeiten soll hier vermieden werden. Vielmehr erfolgt im Falle einer gemeinsam genutzten Ressource jeweils eine explizite Zuordnung zu jeder individuellen Komponente (auch wenn diese vom gleichen Typ ist). Die fragliche Ressorce selbst ist dabei für einen reentranten oder synchronisierenden Zugriff auf sich zuständig.[25] Insbesondere entfällt in dieser Komponentensicht die Möglichkeit synchronisierenden »Konstruktor-Codes« wie ihn die Objektsicht häufig nahelegt, da keine gemeinsamen Konstruktoren existieren. Die analogen, in der Komponentensicht präferierten *Home* bzw. *Factory* Konstrukte stellen im Sinne einer Externalisierung dann wieder individuelle Komponenten dar.

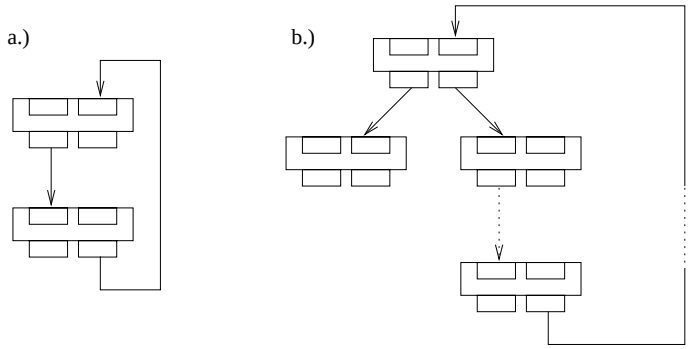
Daneben könnte aber auch versucht werden, die Anwendungsfunktionalität, die den Einsatz bestimmter Komponenten voraussetzt, innerhalb einer einzelnen Komponente zu konzentrieren, statt über mehrere zu verteilen. Die Notwendigkeit einer

[25] Dies schließt nicht die Verwendung zum Beispiel einer externen »Datenmanagementkomponente« zur Verwaltung der nebenläufigen Zugriffe auf die Ressource aus.

Mehrfachverwendung wäre dann nicht mehr gegeben. Hieraus ließe sich auch ein Verfahren zu Identifikation von Komponenten für einen Anwendungsbereich gewinnen. Ein Softwarebaustein wäre genau dann eine Komponente, wenn eine vollständige funktionale Abdeckung eines Domänenkonzeptes (vgl. Seite 90 und Abschnitt 5.3) derart erfolgt, dass keine Mehrfachverwendung von Komponenten mehr auftritt.

Zyklische Abhängigkeiten Ein struktureller Zyklus würde einem mehrfachen Auftreten einer individuellen Komponente in der gesamten Applikationsstruktur entsprechen. Der Strukturgraph eines komponentenbasierten Systems darf dementsprechend keine Zyklen enthalten.[26] Ein funktionaler Zyklus sollte dagegen zugelassen werden (s. Abbildung 2.3), da er in der Programmierpraxis öfter zu beobachten ist. Allerdings muss dann die strukturelle Eindeutigkeit der Nutzung einer Komponente, durch eine explizite Berücksichtigung der *Zeitpunkte* ihrer Nutzung begleitet werden, um auch eine funktionale Eindeutigkeit zu erhalten. Abschnitt 2.3.5 betrachtet entsprechende Möglichkeiten.

Abbildung 2.3
Zyklen



Schnittstellen Bei einer detaillierteren, weniger abstrahierenden Betrachtung bestehen die strukturellen Beziehungen im Modell nicht zwischen den Komponenten als Ganzes, sondern zwischen ihren Schnittstellen. Strukturelle Aspekte umfassen dann die folgenden Bereiche:

- Anzahl der Schnittstellen pro Komponente

[26] Ein Zyklus kann hier als »indirekte« Reflexivität gedeutet werden und ist durch die in der angegebenen Basisrelation geforderte Irreflexivität dann wie gewünscht ausgeschlossen.

- Unterscheidung von Import- und Exportschnittstellen
- Wechselseitige Schnittstellenbeziehungen und komponentenspezifische Zuordnung
- Schnittstelleninhalte
  - Operationelle Schnittstellen
  - Streamschnittstellen
  - Signalschnittstellen

Bezüglich der *Schnittstellenanzahl* pro Komponente ist vor allem die Unterscheidung zwischen $1:1$ und $n:1$ Beziehungen ein wesentlicher Faktor. Mehrere Schnittstellen pro Komponente sind dabei als pragmatischerer Ansatz zu betrachten, der eine logische Untergliederung der Funktionalitäten einer Komponente während ihrer Modellierung erlaubt. Multiple Sichten oder *Rollen* (vgl. a. Abschnitt 4.8) lassen sich implementieren. Durch das Hinzufügen von neuen, verbesserten Schnittstellenversionen zu den bereits bestehenden – und verwendeten – kann eine abwärtskompatible Evolution erreicht werden.[27]

Die Beschränkung auf genau eine Schnittstelle pro Komponente führt hingegen zu einem potenziell klareren, einfacheren Modell. Probleme der Spezifikation bei multiplen Schnittstellen werden umgangen, da keine internen Abhängigkeiten bestehen *können*. Insgesamt geht das hier entworfene Modell daher zwar von multiplen, aber *segmentierten*, d.h. ohne implizite Abhängigkeiten (über das Komponenteninnere) existierende Schnittstellen aus (vgl. a. Kapitel 4). Damit entspricht eine Komponente mit mehreren Schnittstellen dann einem Aggregat mehrerer Komponenten mit Einzelschnittstellen, was zum einen in einem sehr homogenen, zum anderen wie gewünscht einem geschlossenen Komponentenmodell resultiert.

Um zwischen der *bereitgestellten* und der *vorausgesetzten* Funktionalität einer Komponenten explizit unterscheiden zu könnnen, werden deren Schnittstellen in *Import*- und *Exportschnittstellen* differenziert. Exportschnittstellen enthalten dann die bereitgestellte Funktionalität einer Komponente, während

[27] Dies wird in der Literatur häufig betont, ist nach Meinung des Autors, aber erst unter Einhaltung der in dieser Arbeit vorgeschlagenen *Segmentierung* sinnvoll möglich, da ansonsten den Vererbungsanomalien der Objekorientierung analoge Probleme auftreten und die Vorteile einer Komponentensicht in Frage stellen würden.

Importschnittstellen die von ihr *vorausgesetzten* Leistungen untergeordneter Einheiten ausdrücken. Im weiteren Verlauf soll damit also von multiplen, explizit zu jeder Komponente gehörigen Im- und Exportschnittstellen ausgegangen werden.

Verfeinerte Struktur Nachdem statische Beziehungen auf der Granularitätsebene ganzer Komponenten betrachtet wurden, wird im Folgenden eine verfeinerte Sicht eingenommen, um entsprechende Aspekte beim späteren Entwurf von Adaptions- und Klassifikationskonzepten berücksichtigen zu können. Neben der eingehenderen Betrachtung multipler Schnittstellen widmet sich der Abschnitt daher Schnittstelleninhalten.

Multiple Schnittstellen Die Berücksichtigung multipler Export- bzw. Importschnittstellen bietet eine Reihe methodologischer Vorteile. Aus struktureller Sicht ergeben sich dadurch zusätzliche Beziehungsformen, deren semantische Auswirkung für eine Systemstruktur zu prüfen ist. Allgemein können die in Abbildung 2.4 skizzierten Fälle unterschieden werden. In der Abbildung werden Beziehungen zwischen kompatiblen Schnittstellen durch Pfeile angedeutet.

Die weiter oben beschriebene Problematik einer multiplen Komponentenverwendung wiederholt sich offensichtlich analog auf dem verfeinerten Niveau der Schnittstellensicht, nämlich immer dann, wenn es zu einer mehrfachen Bindung verschiedener Schnittstellen zweier Bausteine kommt (in den Fällen c und e). Derartige Situationen können durch mehrfache Instanzen des betroffenen Komponententyps aufgelöst werden, was in der Abbildung jeweils durch alternative Fallbehandlungen mit gestrichelten Komponenten angedeutet wird.

Die Auflösung in einzelne Individuen für Fälle wie e) ist auch eine notwendige Voraussetzung für die Überprüfung der Substituitonsfähigkeit einer Komponente durch eine andere innerhalb einer gegebenen Struktur, da die hier verwendetete segmentierte Schnittstellensicht, eine Substitutionsfähigkeit auch für einzelne Schnittstellen entscheiden kann (vgl. a. Kapitel 4).

Elemente einer Schnittstelle Um die über eine Schnittstelle bereitgestellte Interaktionsfähigkeit zu repräsentieren, sollten auf Modellebene möglichst abstrakte Elemente verwendet werden, die ein Modell in seiner Anwendbarkeit und Aussagekraft nicht unnötig beschränken. Auf derartige Basiselemente sollte sich ein Maximum der praxisüblichen Konstrukte wie Funk-

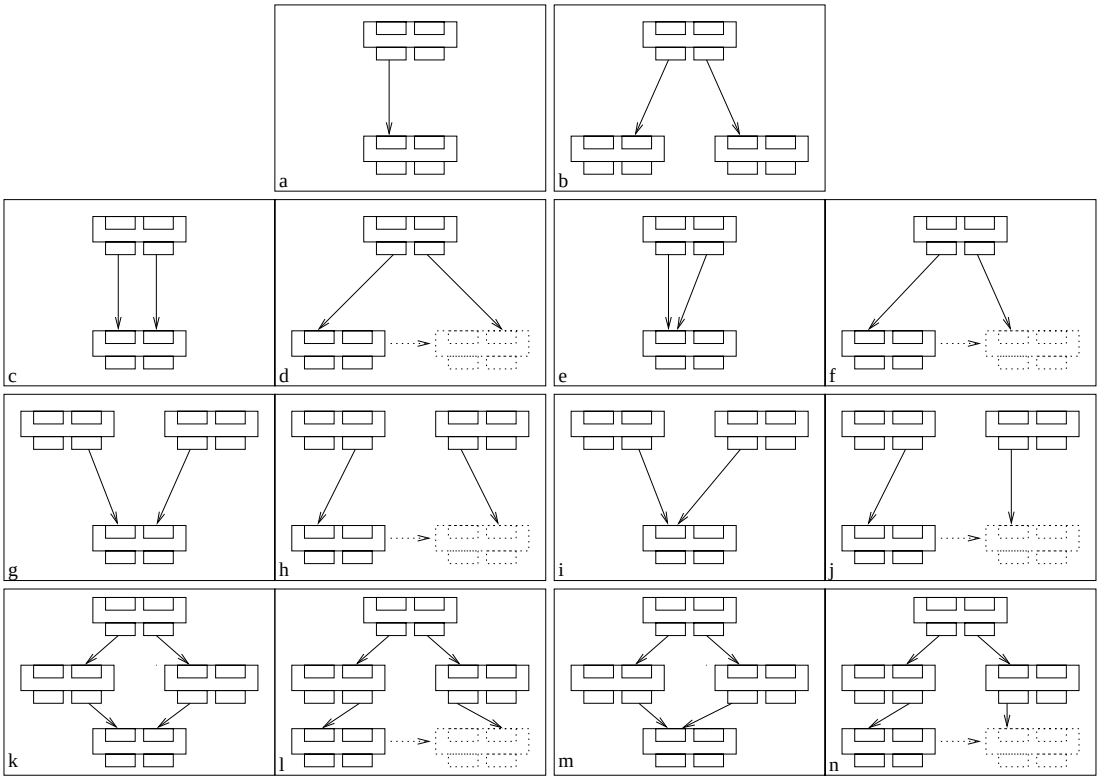


Abbildung 2.4
Mögliche Strukturen bei multiplen Interfaces

tionen, Prozeduren oder Methoden abbilden lassen. Durch die Objektorientierung sehr verbreitet ist zum Beispiel die Kombination aus Methoden, Attributen und Ereignissen.

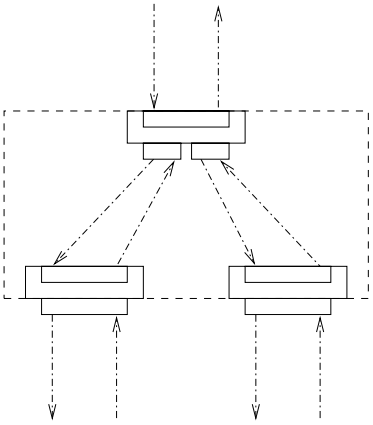
Als potenziell zur Modellierung aller dieser Elemente geeignete Konzepte scheinen die sehr allgemeinen Vorstellungen der *Nachricht* und des *Kanals* vielversprechende Kandidaten zu sein.

- **Kanal** Eine mögliche Abstraktion ist die des *Kanals*. Zwischen zwei Schnittstellen fließen dabei Daten durch eine Menge gerichteter Kanäle. Über Art, Richtung, Reihenfolge und Lebensdauer von Kanälen lassen sich alle oben angedeuteten funktionalen Konstrukte modellieren (s. Abbildung 2.6). Als konkretes Beispiel für die Verwendung von Kanälen in einem Komponentenmodell sei auf Broy [Broy, 1998, Broy, 1996, Broy et al., 1998] verwiesen.

- **Nachricht** Der Begriff der *Nachricht* ist eine weit verbreitete Abstraktion und stellt die Grundlage der Diskussion klassischer Objektmodelle dar. Zwischen den einzel-

nen Schnittstellen kommunizieren die Komponenten in diesem Fall über ein- und ausgehende atomare Einheiten mit festgelegter Bedeutung. So eine Kommunikation findet dabei immer zwischen genau zwei kompatiblen Schnittstellen statt, die ein Import / Export Paar bilden (s. Abbildung 2.5).

Abbildung 2.5
Bidirektionaler Nachrichtenaustausch



Bei der Abbildung parametrisierter Konstrukte[28] auf Nachrichten sind verschiedene Ansätze denkbar. Zunächst kann eine komplexere Sicht des Nachrichtenbegriffs diese als selber mit Parametern versehene Einheiten verstehen. Alternativ können aber auch Parameter ihrerseits wieder als Nachrichten modelliert werden, was die konzeptuelle Simplizität des Modells besser erhält. Dann müssen *geordnete* Folgen von Nachrichten betrachtet werden. Das strukturelle Grundmodell ist dabei durch ein entsprechendes Verhaltensmodell zu ergänzen. Kapitel 4 wird diese Aspekte detailliert darstellen.

Die Wahl von *Nachrichten* zur Modellierung der Kommunikation zwischen Komponenten erlaubt durch ihre Abstraktheit eine Abbildung auf viele konkrete Konstrukte wie Operationen (Funktionen, Prozeduren . . .), Ereignisse, Ausnahmen etc. Insbesondere können sowohl synchrone als auch asynchrone Kommunikationsprozesse beschrieben werden. Die Berücksichtigung von *Daten(-Typen)* innerhalb eines Nachrichtenmodells ist nicht zwingend notwendig, ermöglicht aber eine reichere und ausdrucksstärkere Beschreibung.

[28] beispielsweise Methoden im objektorientierten Sinne

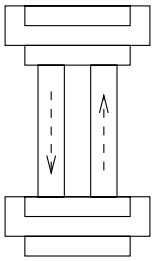


Abbildung 2.6
Kanalabstraktion

2.3.2.2 Verhaltensmodelle

Verhaltensaspekte können auf zwei Ebenen betrachtet werden. Zum einen der Dienstnutzung zwischen Komponenten, zum anderen solche zwischen verschiedenen Diensten einer einzelnen Komponente.[29]

Externes Verhalten Das externe Verhalten beinhaltet allgemeine Eigenschaften der Komponenteninteraktion. In diesem Rahmen muss festgelegt werden, nach welchen grundsätzlichen Regeln Vorgänge zwischen Komponenten ablaufen sollen.

Zunächst lässt sich argumentieren, dass Interaktionen zwischen Komponenten stets von »oben nach unten« eingeleitet werden (vgl. a. Abschnitt 5.7.1). Eine übergeordnete Komponente beginnt in diesem Sinne das kollaborative Zusammenspiel, indem sie den Dienst einer ihrer konstituierenden Teilkomponenten in Anspruch nimmt. Eine Teilkomponente im vorgeschlagenen Modell wird daher andersherum gesagt nie von sich aus Dienstnutzungen übergeordneter Teile initiieren.

Weiterhin kann eine übergeordnete Komponente nur Dienste *unmittelbar benachbarter* Komponenten in Anspruch nehmen, da sie nur mit deren Schnittstellen in Beziehung steht. Funktionalität tiefergelegener Schichten kann somit nur *indirekt* genutzt werden.[30]

[29] Der Begriff des Dienstes wird hier verwendet, um anzudeuten, dass die Gesamtheit oder zumindest eine zusammengefaßte Teilmenge der Einzelfunktionalitäten einer Komponente betrachtet wird. Eine formale Bedeutung trägt der Begriff hier nicht.

[30] Dies minimiert zum einen auch die benötigten »Verbindungen« innerhalb einer komponentenbasierten Anwendung und überträgt zugleich die Dreidimensionalität physich realer Bausteine in die Softwarewelt, was zur Vermeidung eines »n-dimensionalen Spaghetticodes« (vgl. a. die Delegationshierarchien großer objektorientierter Systeme) beiträgt.

Da zwei beliebige Komponenten einer Anwendung immer in eindeutiger gerichteter Beziehung zueinander stehen und eine Dienstnutzung nur in direkter Weise erfolgen kann, sind Komponenteninteraktionen immer eindeutig durch die beteiligten Instanzen geregelt, ohne weitere Seiteneffekte beachten zu müssen.

Der letzte Punkt hat zur Folge, daß insbesondere eine wechselseitige Interaktion zweier Komponenten derselben »Ebene« (s.a. Abschnitt 6.4.3) immer über einen »Mediator« – die jeweilige Toplevel-Komponente – erfolgt. Diese Tatsache hat eine Reihe positiver Auswirkungen. Ein besonderer Vorteil liegt darin, dass die Eigenschaften der Interaktion durch die Mediatorkomponente *explizit* gemacht und somit fassbar werden, anstatt implizit in den beteiligten Bausteinen zu verdeckten Abhängigkeiten zu führen [Pazzi, 1997].

Internes Verhalten Einer der wesentlichsten Punkte bei der Modellierung von Komponentensystemen betrifft die Semantik der Teilbausteine. Diesem Aspekt wird in herkömmlichen Ansätzen bisher nur sehr wenig Beachtung geschenkt. Derartige Systeme beschreiben in der Regel nur statische Aspekte ihrer Bestandteile, wie etwa die Signaturen von Operationen innerhalb von Schnittstellen. Die Semantik dieser Operationen wird dann durch Typisierung eingebracht, die aber in der Regel nur die einzelnen auftretenden Elemente erfasst, nicht aber die Signatur als ganzes. Durch dieses Vorgehen lassen sich weder inhaltliche Bedeutung noch sinnvolle Anwendung der funktionalen Elemente fachnah ableiten.

Dieser Teilaspekt einer Komponente – ihr gemäß den fachlichen Regelungen einer Domäne erfolgender Einsatz – manifestiert sich konkreter in den dynamischen Beziehungen verschiedener Basiselemente des jeweiligen Bausteins. Zu diesen Beziehungen gehören vor allem die kausalen Abhängigkeiten bzw. Ausführungsreihenfolgen der verschiedenen Elemente. Ein Komponentenmodell sollte diese Beziehungen möglichst explizit beinhalten.

Die entsprechende Nutzungssemantik einer Komponente kann dabei stets nur bis zu einem gewissen Grad sinnvoll über ein entsprechendes Verhaltensmodell erfasst werden. Eine weitere Verfeinerung und Aufgliederung der jeweils detailreichsten Ebene eines solchen Modells, ist stets von den gewünschten Einsatzmöglichkeiten des betreffenden Modells abhängig. Als Beispiel hierfür kann auch das konkrete Vorgehen beim Entwurf des Klassifikationskonzeptes in Kapitel 4 dienen.

2.3.2.3 Metamodelle

Komponentenmodelle müssen vielerlei Ansprüchen genügen. Zum einen sollen sie ausdrucksstark und detailliert sein, um die Vorgaben und Anforderungen auf technischem Niveau umsetzen zu können, zum anderen ist Übersichtlichkeit und die Möglichkeit allgemeiner, grober Klassifizierung zur praktischen Handhabbarkeit zu bewahren. Der gegensätzliche Standpunkt dieser Sichten lässt es sinnvoll erscheinen, die Einführung einer abstrakteren Metaebene in Erwägung zu ziehen, die auch eine *modellübergreifende* Betrachtung von Komponenten erlaubt.

Ansätze für entsprechende Metamodelle sollten auf einer ganzheitlichen Komponentensicht von außen beruhen. Das Hauptanliegen ist dabei die pragmatische Klassifizierung von Komponenten in grobe Themenbereiche. Ein entsprechendes Typsystem wäre gegenüber der detailierten, im technischen Sinne kompositionsorientierten Modellsicht wesentlich in seiner Komplexität reduziert. Auf Grund der informalen Natur ist diese Sicht demgegenüber aber nur durch eine explizite Form der Zuweisung von Bausteinen zu Klassen und Benennung derer wechselseitigen Beziehungen denkbar.

Für die Suche einer Komponente stellten die exakten Beschreibungsformen der Modellsicht eine Überspezifikation dar, die eher kontraproduktiv sein dürfte, denn gerade in der Entwurfsphase sollen noch keine exakten Details der Realisierung beachtet werden müssen. Lediglich das Interesse an einer bestimmten allgemeinen Funktionalität besteht, die besser in einem Metamodell beschrieben werden könnte. Dies hat auch die in dieser Arbeit präferierte *Merkmalssicht* (s.a. Abschnitt 5.3) motiviert.

Metamodelle erscheinen ebenfalls geeignet zur Manifestierung abstrakter Anwendungs*architekturen*. Mit ihrer Hilfe könnte – in Analogie zu *Entwurfsmustern* [Gamma et al., 1994] – eine Beschreibung allgemeiner, wiederverwendbarer Entwurfsinformationen von Komponentenstrukturen möglich werden. Ein solches Anwendungsskelett könnte dann mit Instanzen der grob umschriebenen Bausteinarten konkret ausgeprägt werden, wobei durch entsprechende Auswahl aus den relativ großen Klassen[31] doch noch weitgehende Freiheitsgrade resultierten, um zu Applikationen mit individuellem Charakter und Einsatzbereich zu gelangen. Kapitel 5 greift diese Thematik wieder auf.

[31] Eine entsprechende Größe ergibt sich allein schon auf Grund der gefordertenen Variabilität und Flexibilität von Komponenten.

2.3.3 Die Beschreibungsebene

Die Beschreibungsebene von Komponentenarchitekturen beinhaltet als wesentliche Bestandteile Spezifikationen – also Beschreibungen von Eigenschaften –, die möglichst formal gefasst werden und deren Sinn in der Abbildung der informalen Konzepte der oben betrachteten Modellebene besteht, um diese in einem anschließenden Schritt einer automatisierbaren Auswertung und Verarbeitung zuzuführen. Gelingt dieser Übergang vom informalen, konzeptuellen Modell zur konkreten Implementation bruchlos, können Entwicklungsprozesse die frühen, problemnahen und fachbezogenen Phasen der Analyse und des Designs mehr betonen und die wenig problemspezifischen Hürden der konkreten Codierung in den Hintergrund treten lassen. Als extreme Ausprägung dieser Sichtweise steht dann der hier vorgeschlagene Prozess eine generativen Softwarekonstruktion unter vollständiger Abstraktion technischer Gegebenheiten einer Software.

Im weiteren Verlauf werden daher zunächst die methodologischen Einflußfaktoren der Beschreibungsebene betrachtet. Im Anschluß folgen dann detaillierte Betrachtungen der Bereiche Spezifikation, Formalisierung und Typisierung.

2.3.3.1 Einflußfaktoren

Identifikation von Komponentenaggregaten

- Einer **Top-Down Strategie** folgend kann ein Komponentenaggregat rein aus den äußeren Vorgaben an die zu exportierende Funktionalität abgeleitet werden. Dies kann im Extremfall der Wunsch nach einer kompletten Anwendung sein. In der Regel werden die Vorgaben für Komponenten jedoch aus der funktionellen Dekomposition umfangreicherer Fachkonzepte einer Anwendungsdomäne entstehen.

  Im Wechselspiel einer Domänenenanalyse und des zugehörigen Domänenentwurfs (s.a. Abschnitt 5.3 wird dabei nach Vorgabe einer vollständig spezifizierten Exportschnittstelle ein Design abgeleitet, aus dem sich in gleicher Weise komplett spezifizierte Anforderungen für weitere (»Sub-«) Komponenten ergeben, ohne dass diese als explizit implementierte Bausteine in dieser Phase zur Verfügung stehen müssten.

  Anforderungen in Form vorausgesetzter Schnittstellen, ausgedrückt mittels formaler Spezifikationen, können entweder durch Komponenten mit exakt passenden »Gegen-

stücken« erfüllt werden oder durch solche, deren Schnittstellen äquivalent zu den geforderten Spezifikationen sind. Diese Schnittstellenäquivalenzen sowie eventuell notwendige Adapterkomponenten lassen sich aus den formalen Spezifikationen ableiten. Dies stellt die Grundlage des später diskutierten adaptiven und des generativen Konzeptes insgesamt dar.

- Ein pragmatischerer **Bottom-Up-** bzw. **Framework-Ansatz** könnte ebenfalls von einer anzustrebenden Exportfunktionalität ausgehen, diese aber durch ausschließliche Komposition bereits vorhandener Komponenten[32] zu erreichen suchen. Die spezifizierten Import-Funktionalitäten werden dann durch eine Kombination in die (abstraktere) Export-Funktionalität erhoben. Die formalen Spezifikationen von Exportfunktionalität, interner Implementation und einzelnen Importfunktionalitäten können dann dazu verwendet werden, die Korrektheit des so entstandenen Aggregats zu verifizieren. Zu einem späteren Zeitpunkt ist es dann möglich, die Teilkomponenten im Rahmen einer speziellen Konfiguration oder Evolution durch spezifikationsäquivalente Einheiten zu ersetzen. Hier kommen wieder formale Spezifikationen zum Einsatz, welche bei Bedarf auch die Generierung entsprechender Adapter erlauben. Kapitel 3 stellt einen entsprechenden Ansatz vor.

Der in der vorliegenden Arbeit entwickelte generative Ansatz ist nach dieser Einordnung durch eine Top-Down Vorgehensweise in der Design-Phase und ein Bottom-Up Vorgehen in der Konstruktionsphase gekennzeichnet (s. Kapitel 5).

Aufteilung einer Spezifikation

- Als erster Gedanke bietet sich die Verwendung genau einer, bezüglich der Komponente globaler und im Wesen universellen Spezifikation an. Dies entspricht dann der Offenlegung genau solcher Teile, welche in Bezug auf die Gesamtstruktur von globaler Bedeutung sind, was auch in [Büchi & Weck, 1997] vorgeschlagen wird. Ein solche zusammengefasste, globale Repräsentation würde sich speziell bei einer Top-Down-Strategie ergeben, denn aus übergeordneter Sicht ist die in der Regel erfolgende Aufteilung vor-

[32] Dabei ist es konzeptuell ohne Belang, ob die Komponenten tatsächlich bereits existieren oder nur eine ihrer Klassifikation entsprechende Spezifikation ihrer Eigenschaften.

handener Anforderungen auf eine Menge von Teilanforderungen, welche dann durch eine entsprechende Anzahl von Teilkomponenten erfüllt wird, nicht von Bedeutung. Diese Eigenschaft erweist sich aber gleichermaßen auch als Problematik des Ansatzes, denn die Art und Weise einer solchen Zerlegung – d.h., wie zerlegt werden könnte und sollte – bleibt dabei ungeklärt.

- Als komplementäre Form ist eine Menge separater Teilspezifikationen entsprechend unterschiedener Importschnittstellen – also angedachter Teilkomponenten – denkbar. Diese Variante kommt insbesondere dem Buttom-Up Ansatz entgegen. Als offensichtliche Schwäche ergibt sich allerdings sofort das Fehlen einer globalen Sicht – es stellt sich die Frage nach dem korrekten Zusammenspiel der Teilkomponenten im Sinne eines validen Aggregats als gewünschtes Resultat.

Die komplementären Stärken und Schwächen der beiden Spezifikationsvarianten legen den Ansatz einer kombinierten Form nahe. Solche mehrschichtigen Spezifikationen sollten sich aber immer noch an der gewünschten Methodik orientieren und in diesem Sinne einen Schwerpunkt setzen, um eine ausreichende Pragmatik zu gewährleisten.

Modellannahmen Um zu geeigneten Spezifikationsansätzen zu gelangen, muss zum einen festgelegt werden, auf welche Weise die externen Spezifikationen von Komponenten bei deren Entwurf entstehen und zum anderen, wie die Anforderungen einer Komponente an ihre Teilkomponenten und deren Spezifikationen zustande kommen. Antworten auf diese Fragen bestimmen die Struktur von Komponentenspezifikationen. Aus den Überlegungen des letzten Abschnitts lassen sich zusammenfassend zunächst die folgenden Aussagen treffen:

- Die – gewünschte – externe Funktionalität einer Komponente ist initial bekannt. Diese entspricht der später bereitgestellten Dienstleistung des Bausteins.
- Die funktionalen Eigenschaften einer Komponente ergeben sich endgültig erst zum Entwicklungszeitpunkt, da sie von internen Implementationsentscheidungen beeinflußt werden.[33]

[33] Andernfalls könnte vollständig auf Validierungstests verzichtet werden.

- Zumindest die funktionalen Basiselemente – etwa Nachrichtentypen – sind zum Entwurfszeitpunkt bekannt. Dies ist notwendig, um Spezifikationen eindeutig ausdrücken zu können.

- Die Partitionierung der geforderten Funktionalität in disjunkte Mengen, welche den Teilkomponenten eines Aggregats entsprechen, ist in klassischen Entwicklungsweisen nicht notwendigerweise zum Entwurfszeitpunkt bekannt. Hier wird jedoch eine explizite Domänenaufbereitung gefordert, so das auch eine »natürliche« (problemspzifische) Partitionierung vorliegt (vgl. Abschnitt 5.3).

2.3.3.2 Charakter einer Spezifikation

Eine *Spezifikation* drückt in expliziter Form die Eigenschaften des Spezifikationsgegenstandes aus. Der Nutzen von Spezifikationen liegt darin, daß sie eine unmißverständliche Grundlage für die kommunikative Argumentation über Rahmenbedingungen darstellen. Dies gilt zum einen für das Propagieren eines Angebots und zum anderen für das Retrieval einer Nachfrage.

Der folgende Abschnitt untersucht in diesem Zusammenhang Aspekte der *Komponentenspezifikation*. Die zentrale Frage dabei lautet:

> Welche Modellaspekte sollten spezifiziert werden, welche Ausdrucksmittel sind dafür geeignet, und wie sollten diese organisiert bzw. strukturiert werden?

Im Folgenden wird zunächst der *Abstraktionsgrad* von Komponentenspezifikationen hinterfragt, d.h. es wird untersucht, wie detailliert eine entsprechende Beschreibung sein sollte. Im Anschluß folgt dann eine genauere Betrachtung konkreter Inhalte, welche sich in die Schwerpunkte *Struktur*- und *Verhaltensspezifikation* gliedert.

Abstraktionsgrad Zum einen soll die formale Beschreibung einer Spezifikation hinreichend sein, um die Anwendung und Validierung einer Komponente zu ermöglichen, zum anderen soll aber von allen unnötigen Details abstrahiert werden, um keine zusätzlichen Abhängigkeiten zu schaffen. Man kann hier die entsprechenden Extreme der *Black-Box* und *White-Box* Ansätze unterscheiden. Black-Box Ansätze abstrahieren dabei vollkommen von allen internen Details, stellen aber unter Umständen

eine zu sehr vereinfachende Sicht dar.[34] White-Box Ansätze legen jegliche Details unmißverständlich offen, laufen damit aber Gefahr beliebige Abhängigkeiten innerhalb eines Systems zu unterstützen. Büchi schlägt in [Büchi & Weck, 1997] alternativ eine *Grey-Box*-Variante vor, welche neben der abstrakten, externen Sicht eine festzulegende Untermenge der Detailinformationen nach außen sichtbar macht und somit die Möglichkeit bietet, genau solche Interna zu spezifizieren, deren Kenntnis für externe Anwendungen des Bausteins essentiell sind. Als grundsätzliche Möglichkeiten der Spezifikation von Komponenten sieht Büchi die (auch sonst in der Literatur gängigen) Varianten

- Vor- / Nachbedingungen (engl. *pre-/post-conditions*),
- Invarianten und temporale Attribute (Properties),
- algebraische bzw. denotationelle Spezifikationen sowie
- abstrakte Anweisungen (engl. *abstract statements*).

Er schlägt dabei abstrakte Anweisungen – etwa in Form von Spracherweiterungen – als besten Ansatz für Grey-Box Komponenten vor. Diese Abstraktion setzt der *interaktionsorientierte* Ansatz dieser Arbeit fort, indem fachnahe domänenspezifische Interaktionen mit einem von konkreten Programmiersprachen entkoppelten Typmodell vorgeschlagen werden (vgl. a. Kapitel 4).

Komponentenbeziehungen Es stellt sich die Frage, wie und ob die externe Struktur, d.h. Beziehungen von Komponenten innerhalb einer Anwendungsstruktur, durch Spezifikationen beschrieben werden sollen. Beziehungen zwischen Komponenten können den gegenseitigen Bezug auf Strukturelemente – zumeist Schnittstellen – sowie eine Vorgangsbeschreibung der Interaktion beinhalten. In der Literatur finden sich zu dieser Thematik verschiedene Beiträge mit unterschiedlich abstrakten Vorgehensweisen:

Olaffson beschreibt die Beziehungen zwischen Komponenten in einem integrierten Ansatz durch *vorausgesetzte Schnittstellen* (engl. *Required Interfaces*) [Olafsson & Bryan, 1997].

[34] In dieser Arbeit wird der Black-Box Ansatz propagiert mit dem Hinweis auf die Bereitstellung konkreter Spezifikationen als Teil einer Komponente und deren automatisierter Auswertung, sodass die notwendigen Detailinformationen in der Tat vorhanden sind, sich für den Anwender (Softwarekonstrukteur) jedoch dennoch eine gekapselte Sicht mit der Konzentration auf die wesentlichen Komponentenmerkmale ergibt.

Murer schlägt dagegen zur Spezifikation der einzelnen Aspekte *Schichten verschiedener Interoperabilitätsinformation* [Murer et al., 1997] vor. Als dritten Ansatz verwenden Mens und Hondt eine separate Beschreibung der von einer Komponente genutzten Funktionalität durch *Reuse Contracts*, welche von konkreten Schnittstellen abstrahieren und so auch deren potenzielle Evolution einbeziehen [Mens et al., 1997, Hondt et al., 1997]. Es sei darauf hingewiesen, dass alle diese Ansätze zunächst theoretischer Natur sind und bislang nicht in die Entwicklungspraxis übertragen wurden. Aber eben diese Praktikabilität ist das Ansinnen des Autors, sodass als gangbarer Ansatz im Folgenden insbesondere das Prinzip der vorausgesetzen Importschnittstellen hervorgehoben werden soll.

Verhaltensspezifikation Bei der Beschreibung (externer) Komponentenfunktionalität lassen sich nun verschiedene Verhaltensbereiche identifizieren, in die sich das Gesamtverhalten aufteilt. Zunächst kann grob zwischen bereitgestellter Export- und vorausgesetzter Importfunktionalität unterschieden werden. Diese beiden Oberbereiche untergliedern sich auf gleiche Weise in die folgenden drei Unterkategorien verschiedener Verhaltensaspekte:

- Allgemeine, zentrale Komponenteneigenschaften
- Einzelne Organisationseinheiten (z.B. *export bzw. required Interfaces*)
- Wechselseitige Abhängigkeiten der Organisationseinheiten (zum Beispiel *Synchronisation*)

Die Thematik der Organisation von Teilgebieten der Komponentenbeschreibung in Spezifikationsstrukturen wird weiter unten noch genauer behandelt.

Eigenschaften etwa von Synchronisationsbedingungen (zum Beispiel »Critical Sections« bzw. »exklusive Ressourcen«) oder einer Anwendungslogik (zum Beispiel `not read_file until open_file`) sind formal schwer zu fassen. Daher muss eine Beschränkung auf Teilaspekte erfolgen. Als sinnvolle dynamische bzw. semantische Spezifikationsbestandteile bieten sich vor allem temporale bzw. kausale (Ursache-Wirkung Reihenfolgen) Beziehungen an. Solche Beziehungen könnten in Bezug auf strukturelle Bestandteile formuliert werden, wobei sich hier besonders die grundlegenden funktionalen Basislemente der Komponentenschnittstellen anbieten. Auf diese Weise lassen sich dann

beispielsweise die fachlich sinnvollen Abfolgen von Methodenaufrufen beschreiben. Um komplexere Verhaltensaspekte zu behandeln, kann wie im Folgenden ein Zustandsbegriff eingeführt werden.

Die Rolle von Zuständen Der Begriff des Zustandes spielt eine wichtige Rolle zur semantisch reichen Beschreibung einer Komponentencharakteristik, da er eine nützliche Abstraktion zur Spezifikation von Verhaltensweisen darstellt. Zustände sind in diesem Sinne Schnappschüsse der dynamisch veränderlichen Eigenschaften eines Bausteins. Eine Verhaltensweise kann dann durch ihre Wirkung auf diese veränderbaren Eigenschaftswerte dargestellt werden, indem der Zustand vor und der Zustand nach jeder Aktivität angegeben wird. Der Verhaltensbegriff dieser Arbeit abstrahiert jedoch von diesen Detaillierungsgrad[35] und stellt zur Verhaltensdefinition die dabei auftretenden Aktivitäten (Interaktionen) selbst in den Vordergrund.

Bei der Rolle von Zuständen innerhalb einer Komponentenspezifikation können zwei Fälle unterschieden werden. Zum einen können Zustände implizit integriert sein, falls ein Formalismus genutzt wird, der grundsätzlich auf Zuständen beruht (zum Beispiel Automatenmodelle). Hierbei müssen oft zwanghaft Zustände bestimmt werden, die keine direkte Beziehung zum betreffenden fachlichen Ablauf aufweisen. Zum anderen kann auch eine explizite Einführung von Zuständen erfolgen, die direkt den in der betrachteten Domäne beobachtbaren Zuständen entsprechen.[36] Diese sind dann der verständlicheren, effektiveren Spezifikation hilfreich, ohne dass dazu ein Zwang durch den zugrundeliegenden Formalismus bestünde.

Spezifikation nicht-funktionaler Eigenschafen Nicht alle relevanten Eigenschaften von Komponenten sind quantitativer Natur. Wichtige Faktoren wie Performanz, Wiederverwendbarkeit oder Verläßlichkeit sind zunächst qualitative Begriffe. So-

[35] In der derzeitigen Entwicklungspraxis sind entsprechende Vor/Nachbedingungen – so sie denn überhaupt eingesetzt werden – in der Regel mit programmtechnischen Konstrukten assoziiert, was dem Komponentengedanken widerspäche.

[36] Die Spezifikation betrifft bei Berücksichtigung dieser Zustände dann einzelne Komponentenindividuen. Die vorliegende Arbeit fokussiert allerdings auf ein Klassifikationskonzept, sodass die instanzlichen, zustandsbezogenen Fragestellungen auf aktive Laufzeitkomponenten verlagert werden (vgl. a. Abschnitt 3.10).

lange für diese keine expliziten Metriken[37] vorliegen, sollten solche Größen zumindest informativ in eine Komponentenspezifikation einfließen können, um damit ebenfalls im Zugriff klassifizierender und automatisierende Ansätze zu liegen. Derzeit existieren noch wenige Untersuchungen und konkrete Lösungen zu diesem Problem. Zumindest eine in diesem Zusammenhang erwähnenswerte Arbeit ist der *NoFun-Ansatz*, der in [Franch, 1997] beschrieben wird. Im Rahmen der vorliegenden Arbeit werden nicht-funktionale Eigenschaften durch Informationen in den entsprechenden Komponentendeskriptoren informal[38] erfasst und bereitgestellt (vgl. Kapitel 5 und 6).

Spezifikationsstruktur Nachdem die Inhalte einer Spezifikation umrissen wurden, ist im nächsten Schritt deren Struktur zu untersuchen. Zunächst sei dabei vorausgesetzt, dass eine Spezifikation in separater Form gemeinsam mit dem durch sie beschrieben Baustein vorliegt, und nicht etwa eine globale Beschreibung der Gesamtstruktur bzw. Anwendung alle komponentenbezogenen Teilaspekte zentral integriert.[39] Dies entspricht auch der derzeit gängigen Sicht der Entwicklungspraxis des Komponentenparadigmas, das von der Verarbeitung vollständiger Komponenten*archive* ausgeht, die jeweils alle für die Verwendung einer Komponente relevanten Informationen enthalten.[40] Die hier zu betrachtenden Spezifikationen stehen also immer in direktem Zusammenhang zu einem einzelnen korrespondierenden Baustein bzw. Bausteintyp.

Im Folgenden wird nun die strukturelle Organisation solcher Beschreibungen betrachtet, wobei als erstes der enge Zusammenhang mit dem Schnittstellenbegriff disktutiert wird. Im Anschluß an Überlegungen zu integrierten Zentralspezifikationen endet der Abschnitt mit dem Gedanken einer externen Spezifikationsverteilung.

Schnittstellenspezifikation Die Struktur von Komponentenspezifikationen steht in engem Zusammenhang mit dem Be-

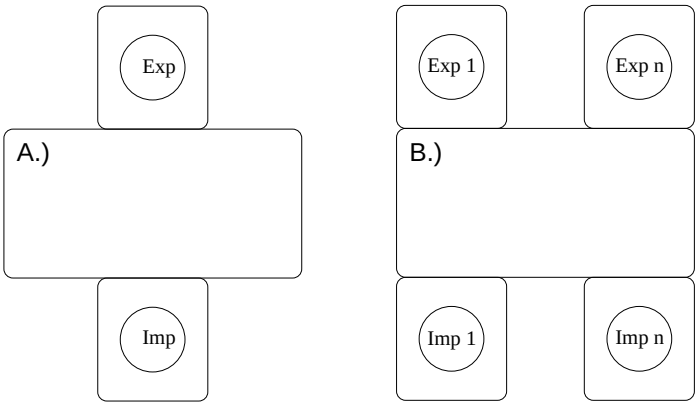
[37] Solche sind nicht Gegenstand der vorliegenden Arbeit, wurden allerdings begleitend zu ihr auch betrachtet (vgl. etwa [Tetau, 1998]).

[38] In wie weit der dabei vorausgesetzte Einsatz einer domänenspezifisch standardisierten Terminologie (Fachvokabular) einem *formalen* Charakter entspricht sei zur Diskussion gestellt.

[39] Dies entspricht der in Abschnitt 2.2 angesprochenen Dezentralisierung des Wissen.

[40] Diese Sichtweise hat in der Praxis zum Konzept der *Deskriptoren* geführt (vgl. Abschnitt 5.9).

Abbildung 2.7
Schnittstellenspezifikation



griff der Schnittstelle. Auf Modellebene manifestieren sich Komponenten nach aussen grundsätzlich über solche Schnittstellen als einzige definierte Bezugspunkte. Die Schnittstellen müssen daher in diesem Bild gleichermaßen Träger – oder zumindest Zugangspunkt – und eigentlicher Gegenstand der Spezifikationen sein. In diesem Sinne stellt sich die Spezifikation einer Komponente als Schnittstellenspezifikation dar. Abbildung 2.7 illustriert den Fall von Schnittstellen als jeweilige Träger ihrer eigenen separaten Spezifikation.

In Abschnitt 2.3.2.1 wurden zwei mögliche Varianten der Schnittstellenmenge pro Komponente diskutiert. Diese Varianten wirken sich nun auch auf die Struktur einer Schnittstellenspezifikation aus.

Der Ansatz mehrerer Export- und / oder Importschnittstellen pro Komponente stellt wie gesehen einen äußerst pragmatischen Ansatz mit guten Gliederungsmöglichkeiten und der Unterstützung multipler Sichten/Rollen sowie abwärtskompatibler Evolution dar.

Aus Spezifikationssicht tritt ein wichtiger Aspekt hinzu: Verschiedene Schnittstellen sollten nicht grundsätzlich unabhängig voneinander sein, da sonst außer organisatorischen Aspekten kein Vorteil gegenüber mehreren Komponenten mit jeweils einer Schnittstelle besteht. Bei mehreren Schnittstellen ist somit die wechselseitige Synchronisation der Funktionalitäten zu beachten.

Es muss an dieser Stelle betont werden, dass diese Diskussion hier benötigt wird, obwohl bereits das segmentierte Modell dieser Arbeit betont wurde (s.a. Abschnitt 2.1.1.1 und 2.3.2.1). Diese Segmentierung soll jedoch nur für die fachnahe, pro-

blembezogene Betrachtung einer Komponente gelten (vgl. a. das Entwurfsmuster *Fassade* [Gamma et al., 1994]). Die Möglichkeit *technischer* Implementationen mit entsprechenden Abhängigkeiten soll aus Erwägungen der Praktikabiltität nicht von vornherein ausgeschlossen werden. Die Darstellung des mehrstufigen Typmodells in Kapitel 4 wird diese Diskussion deutlicher werden lassen.

Als Beispiel möge folgende Situation dienen: Eine technische Komponente verfüge über verschiedene Schnittstellen für den Betrieb (`B`) und die Wartung (`W`). Diese können genau dann nicht unabhängig voneinander agieren, wenn Betrieb und Wartung sich wechselseitig ausschließen. In diesem Fall müssen die entsprechenden Dienste etwa in folgender Weise schnittstellenübergreifend synchronisiert werden:[41]

```
B:startWorking → B:stopWorking →
W:startService → W:stopService →
B:startWorking ...
```

Für die Konzeption einer diesbezüglichen Spezifikation ist die Frage von Bedeutung, ob sich die Synchronisation der multiplen Exportschnittstellen auf das externe Verhalten von Komponenten auswirken kann. Wenn ja, müssen die Synchronisationsbedingungen in die Modellspezifikationen integriert werden. Dies könnte durch einen zentralen Synchronisationsteil erfolgen, wie bei der Komponente D.) in Abbildung 2.8 veranschaulicht.

Zu bedenken ist weiterhin der Fall multipler Komponentennutzung, bei dem verschiedene Exportschnittstellen von unterschiedlichen anderen Komponenten aus genutzt werden. Es ist dann zu beachten, dass die theoretische Lebendigkeit der Komponente bei Inter-Export-Beziehungen (Komponentennutzung durch verschiedenen Strukturen/Anwendungen) gewährleistet werden muss. Eventuell müßten Inter-Export-Beziehungen dieser Art in einer restriktiveren Form vorliegen.

Als zweite Möglichkeit wurde genau eine Export- und Importschnittstelle pro Komponente in Erwägung gezogen. Bei diesem Ansatz treten die Probleme der Spezifikation bei multiplen Schnittstellen naturgemäß nicht auf. Er führt dementsprechend

[41] Ein Automatenmodell wie es in den nachfolgenden Kapiteln zu Grunde gelegt werden soll wäre hier nicht ausreichend, da es einen Protokollautomaten je Schnittstelle vorsieht und damit nicht schnittstellenübergreifend anwendbar ist. Die Lösung liegt in der mehrstufigen Konzeption des in Kapitel 4 vorgestellten Typmodells, sodass in der oberen Ebene des Modells vom Schnittstellenbegriff abstrahiert werden kann.

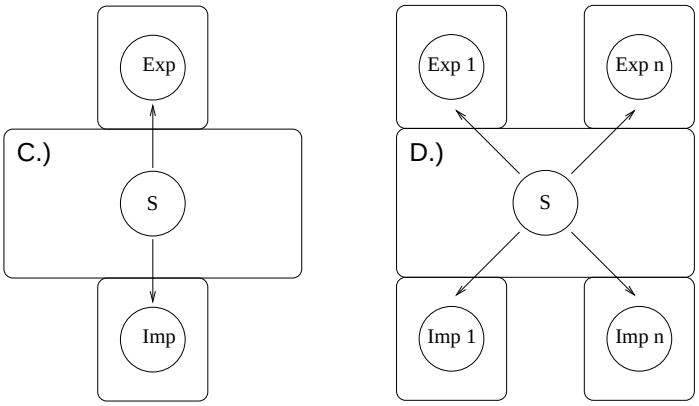


Abbildung 2.8
Synchronisierte Einzel- und Multischnittstellen-spezifikation

zu unkomplizierteren Architekturen. Trotzdem sollte auch in diesem Fall nicht auf einen Synchronisationsteil verzichtet werden. Dessen Aufgabe ist es, hier die Beziehungen zwischen dem Import- und dem Exportteil explizit werden zu lassen (Abb. 2.8 Komponente C.)). Insofern stellt eine entsprechende Spezifikation dann auch eine sinnvolle Ergäzung der segmentierenden Sichtweise des in Kapitel 4 vorgestellen Klassifikationsansatzes dar.

Zentrale Spezifikationen Komponentenspezifikationen enthalten wie gesehen Beschreibungen von Schnittstellen. Es liegt daher nahe, für die einzelnen Schnittstellen eines Bausteins Spezifikationen aufzustellen, welche dann zusammen eine disjunkte Menge separater Teilbeschreibungen ergeben. Wie oben ausgeführt, fehlen bei diesem Ansatz die wechselseitigen Abhängigkeiten einzelner Schnittstellen, was besonders bei multiplen Schnittstellen zu unangemessenen Resultaten führt. Es sind also zumindest zusätzliche Beschreibungen für die Synchronisation abhängiger Teilschnittstellen zu ergänzen. An diesem Punkt stellt sich die Frage, ob Komponentenspezifikationen überhaupt einen Bezug zum Schnittstellenbegriff benötigen, oder ob eine einzelne zentrale Gesamtspezifikation nicht alle wesentlichen Merkmale auszudrücken vermag.

Bestärkt wird obige Frage durch die Tatsache, dass ein abstraktes, formales Modell prinzipiell vollkommen auf Schnittstellen verzichten kann und sich auf die ausschließliche Betrachtung einer Komponente als Ganzes und ihrer kommunikativen Basiselemente (zum Beispiel Nachrichten) beschränken lässt. Eine Deklaration von Schnittstellen als Teilmengen der gesam-

ten Nachrichtenmenge ist aus dieser Sicht rein organisatorischer Natur. Die Spezifikation von semantischen Abhängigkeiten zwischen verschiedenen Schnittstellen – konkreter deren Kommunikationseinheiten – ist nicht auf diese Organisationsstruktur angewiesen.

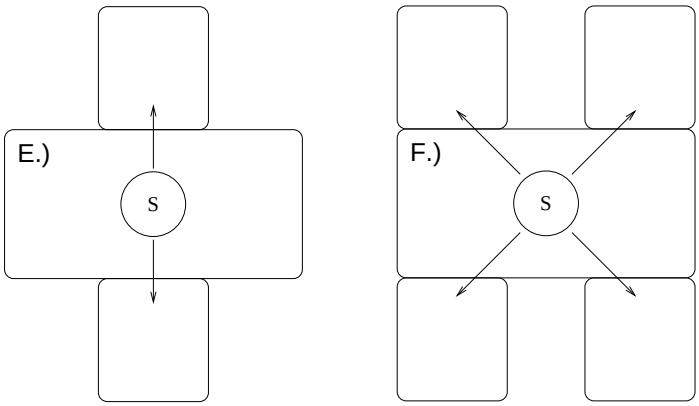


Abbildung 2.9
Zentralspezifikation

So eine Zentralspezifikation (siehe Abbildung 2.9) kann die Komponentenbeschreibung ferner in verschiedener Hinsicht bereichern. Sie ist etwa in der Lage, Aspekte zu berücksichtigen, welche sich aus der internen Implementation des Bausteins ergeben und nicht in direktem Zusammenhang zu einzelnen Schnittstellen stehen, oder bei denen diese Beziehung nicht direkt explizit ausgedrückt werden soll. Weiterhin bietet sie eine umfassendere Sicht, mit der die schnittstellenübergreifenden Zusammenhänge übersichtlicher und eindeutiger formuliert werden können.

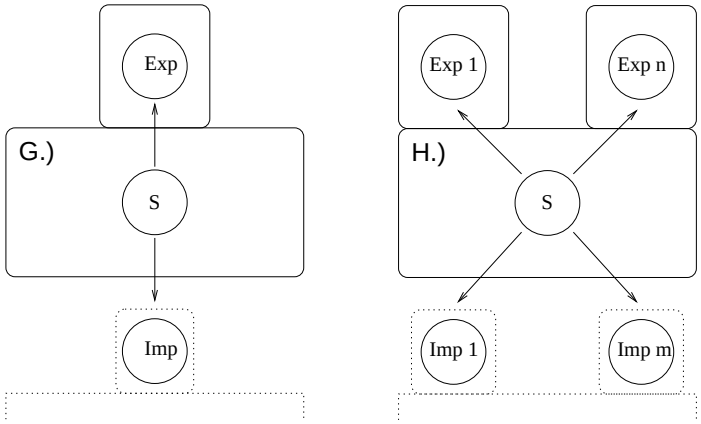
Ein problematischer Aspekt der Zentralspezifikation liegt hingegen darin, dass beim vorgeschlagenen Komponentenmodell wegen der segmentierten Schnittstellensicht generell Teilspezifikationen in Bezug auf einzelne Schnittstellen notwendig sind. Da eine Komponentenkomposition jeweils bestimmte einzelne Schnittstellen nutzt, ist ein Abgleich mit genau deren Eigenschaften notwendig. Teilspezifikationen müssen also entweder abgeleitet werden, oder ein Spezifikationsabgleich muss immer generell über die Zentralspezifikation als Ganzes geschehen. Beide Varianten bedeuten zusätzlichen Aufwand.

Auslagerung von Anforderungspezifikationen Eine Möglichkeit, externe Beziehungen von Komponenten auszudrücken, ist – wie gesehen – die Spezifikation von geforderten Eigenschaf-

ten etwa mittels Import- bzw. vorausgesetzten Schnittstellen. Bezüglich der Zuordnung dieser Spezifikationen zu Komponenten sind nun verschiedene Möglichkeiten zu unterscheiden.

Zunächst kann ein Aggregat mit Anforderungen an seine konstituierenden Teilkomponenten deren Spezifikationen exklusiv beinhalten. Es existieren dann zwei separate Spezifikationen von zum einen geforderter Import- und zum anderen erbrachter Export-Funktionalität. Die Interaktions- bzw. Kompositionsfähigkeit der Partner kann dann mittels Abgleich der Spezifikationen geprüft werden.

Abbildung 2.10
Externe Importspezifikation



Als zweite Alternative ist ein referentieller, verteilter Ansatz denkbar, bei dem das Aggregat seine Anforderungen jeweils mittels Verweis auf die Spezifikation einer konstituierenden Teilkomponente ausdrückt. Eine solche Referenz könnte entweder direkt erfolgen, wobei der entsprechende Teilbaustein – in diesem Fall natürlich im Sinne einer Bottom-Up Strategie (2.3.3.1) – initial existieren muss, oder auf indirekte Weise etwa eine abstrakte externe Typbeschreibung nutzen. Abbildung 2.10 veranschaulicht diese Externalisierung von Anforderungsspezifikationen für den Fall einer direkten Referenz

2.3.3.3 Formalisierung

Für die Formalisierung der bislang diskutierten Ansätze bieten sich verschiedene Alternativen an. Neben klassischen Automatenmodellen scheinen vor allem logikbasierte Ansätze erfolgversprechende Wege zu eröffnen, da sie in der Literatur entsprechend gut verstanden und untersucht sind.

Im Folgenden wird zunächst die Verwendung formaler Theorien im Zusammenhang mit Komponentenstrukturen kritisch hinterfragt und schließlich gerechtfertigt. Im Anschluß folgen dann Überlegungen zu möglichen Formalisierungsansätzen, die sich wieder grob in strukturelle und verhaltensbasierte Bereiche gliedern. Innerhalb dieser Bereiche wird jeweils ein spezieller Ansatz herausgearbeitet.

Warum Formalismen? Bevor im Anschluß konkrete Möglichkeiten der Formalisierung betrachtet werden, sollen zunächst Fragen nach zugedachter Rolle, gewünschter Wirkung und potenziellen negativen Begleiterscheinungen formaler Methoden im Komponentenparadigma zur Sprache kommen.

Für die Anwendung von Formalismen spricht die Tatsache, dass diese zur beweisbaren Korrektheit von Systemen bezüglich verschiedener, alternativer Aspekte führen und damit die Voraussetzungen für komplette und gleichzeitig gesicherte Abstraktionen schaffen. Dadurch kann ein Retrieval von beweisbar korrekten Komponenten unter weniger strengen Recherchebedingungen für informalere Modelle stattfinden, wobei die Möglichkeiten von Teilspezifikationen bzw -verifikationen und deren Kombination besteht. Im Anschluß kann eine gesicherte, automatisierbare Komposition erfolgen, welche in direkter Weise zu neuen, komplett spezifizierten Bausteinen führt und so auch Wiederverwendbarkeit fördert. Als positiver Nebeneffekt führt der Vorgang einer formalen Beschreibung zu einem besseren Systemverständnis. Des weiteren stellen Formalismen eine Basis für Standardisierungen als Voraussetzung für offene Märkte dar.

Den genannten offensichtlichen Vorteilen stehen eine Reihe möglicher Nachteile formaler Methoden gegenüber. Allen voran ist dabei das Akzeptanzproblem zu nennen, welches den Einsatz entsprechender Ansätze häufig verhindert. Der Widerstand entsteht dabei aus dem unvorteilhaften, oft als unangenehm und lästig empfundenen formalen Entwicklungsprozess von Software, welcher mit dem Einsatz entsprechender Methoden einhergeht. Unter ungünstigen Umständen kann dabei der einhergehende informale Entwicklungsprozeß behindert werden. Nicht zu unterschätzen ist daneben der prinzipielle rechnerische Aufwand von deduktivem Retrieval und Beweisen.

Formalismen dienen im allgemeinen dazu, exakte Aussagen über Eigenschaften von Systemen machen zu können. Der wohl wichtigste Aspekt dabei ist, daß mit Formalismen die Korrektheit eines durch sie spezifizierten Systems bewiesen werden kann. Im Gegensatz dazu steht die Alternative der Verifikation von Syste-

men mit Hilfe von Testreihen und Simulationen[42]. Beide Ansätze haben ihre spezifischen Vor- und Nachteile, weshalb keiner ganzheitlich zu überzeugen vermag. Wo auf der einen Seite die vollständige rationale Formalisierung zu untragbarem Aufwand führt, können auf der anderen Seite empirische Tests in sensitiven Bereichen nicht zur notwendigen, vollkommenen Sicherheit führen (vgl. etwa den Ariane Unfall [Lions, 1996]). Als möglicher Ausweg könnte ein Kompromiß angestrebt werden, welcher die Ansätze in ergänzender Weise vereint.

Trotz der ernstzunehmenden Schwierigkeiten steht die letztendliche Vorteilhaftigkeit einer Berücksichtigung formaler Methoden im Komponentenparadigma außer Frage. Es muss jedoch als wesentliche Voraussetzung deren praxisgerechte Anwendung gefordert werden. Dabei ist ebenfalls zu fordern, dass die zur Anwendung kommenden Methoden in einem Rahmen *menschlich* überschaubarer Komplexität verbleiben, der zu einem möglichst einfachen Verständnis beiträgt. In jedem Fall sollte dafür eine automatisierte Unterstützung durch *Systemwerkzeuge* integriert werden, die den Umgang mit entsprechenden Formalismen verbergen. Diese sollten unter anderem auch eine Wiederverwendung von Spezifikationen bzw. Spezifikationsmustern gestatten und begünstigen, um den schwierigen, komplexen Spezifikationsprozess effektiver zu gestalten.

Strukturelle Aspekte Die Struktur einer Komponente drückt sich in der Beschreibung ihrer Schnittstelle(n) aus, welche lediglich Strukturaspekte der Komponenten in Form von herkömmlichen überwiegend operationalen Schnittstellenbeschreibungen mit den darin enthaltenen Signaturen umfassen. Dies wird meist durch syntaktische Elemente einer entsprechenden *Beschreibungssprache* erreicht. Solche Schnittstellenbeschreibungssprachen (*Interface Definition Languages IDL*) führen dabei zu einer Auslagerung der Strukturinformationen aus den eigentlichen Artefakten – den Komponenten –, welche dann allesamt als separate Teile vorliegen. Allgemein lässt sich sagen, daß IDL gleichermaßen weit verbreitet wie ausgereift sind und im allgemeinen auch für Komponentenansätze befriedigende Lösungen darstellen.

[42] Es sei bemerkt, daß diese unterschiedlichen Ansätze in direkter Analogie zu den konträren philosophischen Strömungen von Rationalismus und Empirik [Störig, 1998] stehen. Sie wurden im 18. Jahrhundert von Immanuel Kant zu einer einheitlichen Sicht vereinigt, was als Denkanstoß für analoge Überlegungen innerhalb der Informatik dienen könnte.

Statische Strukturbeziehungen zwischen Komponenten können durch solche Beschreibungsformen automatisiert verifiziert werden. Bekannte Schwachstelle struktureller Spezifikationsformen ist, wie schon früher erwähnt, das Fehlen eindeutiger, semantisch reicher Verhaltensaspekte. So ist es möglich, dass strukturgleiche Komponenten ein voneinander abweichendes Verhalten zeigen, und eine durch die Struktur implizierte scheinbare Konformität bei Berücksichtigung der Verhaltenssemantik keineswegs mehr vorhanden ist. Für Spezifikationen innerhalb eines Komponentenmodells folgt hieraus, dass strukturelle Beschreibungen in jedem Fall durch Verhaltensbeschreibungen ergänzt werden sollten. Auch praktische Bemühungen wie etwa im Rahmen des CORBA Component Modells der OMG zeigen entsprechendes Bemühen, etwa mit erweiterten IDLs wie der CIDL [OMG, 1999].

Ein spezieller Nachteil der angesprochenen IDL ist die relative Starrheit der resultierenden Spezifikationen. Strukturelle Eigenschaften werden direkt und absolut kodiert, weshalb spätere Änderungen zu Problemen führen können. Als flexibler Ansatz für dynamischere Spezifikationen struktureller (und weiterer) Aspekte könnte ein auf der *Feature-Logik* basierendes Konzept dienen.

Feature-Logik Feature-Logik ist eine typisierte Logik, die bei der Formalisierung von flexiblen (Struktur-) Merkmalen einer Komponente hilfreich sein könnte. Die Terme der Feature-Logik beschreiben Objektmengen[43] mit bestimmten Eigenschaften (engl. *Features*) sowie deren Beziehungen. Diese Terme können dazu verwendet werden, Objekte – also auch Komponenten sowie deren konstituierende strukturelle Einheiten – mit bestimmten Eigenschaften dynamisch zu selektieren und einer weiteren Verarbeitung zuzuführen und damit zu einer Klassifikation beitragen. Der Ansatz wird in Abschnitt 2.3.6 eingehender untersucht.

Formale Semantik Bei der Auswahl geeigneter Formalismen liegt ein wesentliches Kriterium – wie schon mehrfach angesprochen – in der Fähigkeit, semantische Eigenschaften adäquat darstellen zu können. Formale Systeme werden unter diesem Gesichtspunkt allgemein in drei Gruppen eingeteilt:

[43] Der Objektbegriff ist hier allgemeiner Natur, nicht etwa auf den Begriff der objektorientieren Programmierung bezogen.

- **Algebraische (denotationale) Semantik** – Denotationale Semantiken bedienen sich streng mathematischer Modelle. Das Verhalten von Operationen einer Komponente würde dann durch algebraische Formeln ausgedrückt. Ein Beispiel für eine algebraische Semantik im Kontext von Komponentenmodellen ist der Ansatz von Broy [Broy, 1996, Broy, 1998]. Dort werden Schnittstellen durch typisierte Kanäle beschrieben, in denen zeitlich getaktete Ströme (*timed streams*) in Komponenten hinein- bzw. hinausfließen. Semantisch werden die Komponenten durch eine Menge von Funktionen zwischen den Input- und Output-Kanälen spezifiziert. Mit diesem Modell lassen sich dann auch Komposition und Verfeinerung von Komponenten sowie beliebige Mischformen daraus darstellen.

- **Axiomatische Semantik** – Bei dieser Gruppe von Formalismen erfolgt eine Deklaration semantischer Aspekte durch Axiomensysteme, welchen das betrachtete System (zum Beispiel eine Komponente) genügen muss. Prominentes Beispiel hierfür sind Pre- und Postconditions. Die semantischen Constraints temporaler Logik fallen ebenso in dieses Gebiet. Axiomatische Semantik erlaubt die Validierung von Systemen bezüglich ihrer Spezifikationskonformität.

- **Operationale Semantik** – Bei operationalen Ansätzen erfolgt die Beschreibung der Semantik eines Systems durch das korrespondierende Verhalten einer grundlegenden Basismaschine. Genauer werden die Zustandsübergänge eines solchen Referenzsystems herangezogen. Diese drücken per Definition eindeutige und von ihrer Bedeutung her klar verständliche Verhaltensaspekte aus. Etwas pragmatischer kann eine einfache, klare und wohlverstandene Programmier-, Pseudo- oder Meta- Sprache verwendet werden, um die Semantik komplizierter, fremdartiger oder zu vergleichender Systeme in eine bekannte, begreifbare Form zu überführen.

Für Komponentenarchitekturen eignen sich besonders axiomatische und operationale Semantiken, da sie pragmatische Lösungen erlauben.[44] Axiomatische Zusicherungen haben sich bei-

[44] Der mathematisch funktionale Charakter denotationaler Semantiken wird den Spezifikationsanforderungen real existierender Softwaresysteme mit ihrer Zustandsbehaftung, ihren Seiteneffekten und etwiger Ausnahmebehandlung weit weniger gerecht, bzw. führt beim Versuch ei-

spielsweise im Bereich der Programmiersprachen – etwa in Eiffel – bewährt [Meyer, 1988]. Schnittstellen von Komponenten werden dabei durch zusätzliche Anforderungen ergänzt, welche die Erwartungen und Versprechen der beteiligten Interaktionspartner ausdrücken.

Büchi schlägt wie oben bereits beschrieben hingegen speziell für Komponentenspezifikationen eher abstrakte Anweisungen aus dem Bereich der operationalen Systeme vor [Büchi & Weck, 1997].

Der Ansatz dieser Arbeit orientiert sich ebenfalls an einer operationalen Sicht, die der beabsichtigten *Interaktionsorientierung* am nächsten kommt, ergänzt diese allerdings durch einen axiomatischen Ansatz mit Hilfe spezieller Randbedingungen (vgl. Abschnitt 4.3).

Der zeitliche Aspekt Um die Aktivitäten von Komponenten im Rahmen ihrer Interaktion zu erfassen, sind die kausalen Beziehungen bzw. zeitlichen Abläufe zwischen deren grundlegenden funktionalen Struktureinheiten von Interesse, die im Folgenden auch als *zeitlicher* oder *temporaler Aspekt* bezeichnet werden sollen.

Endliche Automaten Um solche temporalen Aspekte auszudrücken, bedienen sich Ansätze aus verschiedenen Bereichen oft dem mathematischen Modell des *endlichen Automaten (EA)* [Hopcroft & Ullman, 1993]. Die relative Häufigkeit dieser Lösung hat zwei wesentliche Gründe: Zum einen sind EA leicht verständlich, zum anderen einfach durch Zustandsdiagramme darzustellen und effektive Algorithmen existieren zur konkreten Realisierung. Da die vorliegende Arbeit auf die Unterstützung und Etablierung des Gesamtprozesses einer generativen Softwarekonstruktion abzielt und weniger auf die Betrachtung detaillierter Teilkonzepte, werden solche »einfachen« Konzepte zu Gunsten der Konzentration auf das Gesamtkonzept bevorzugt betrachtet[45]

Die Verwendung von Automatenmodellen erfolgt im allgemeinen nach folgendem Schema [Arapis, 1995]: Ein EA A_K wird

ner vollständigen Abbildung zu ausgesprochen komplexen Formelsystemen. Entsprechend fordert etwa [Jackson & Wing, 1996] leichtgewichtige Formalismen.

[45] Denkbar wäre zum Beispiel auch der Einsatz von Petri-Netzen zur Spezifikation, was zwar zur Steigerung der Ausdrucksmächtigkeit geführt, aber eben wenig zur Gesamtkonzeption beigetragen und den Implementationsaufwand unnötig erhöht hätte.

verwendet, um den temporalen Aspekt einer Ausprägung (etwa eines Objektes oder einer Komponente) des Typs T zu modellieren. Transitionen des EA A_K werden nun mit Operationen beschriftet, welche von Instanzen des Typs T ausgeführt werden sollen. Die Zustände des EA korrespondieren dann mit den Zuständen der Instanz. Eine Transition von A_K, welche mit p beschriftet ist und zwischen Zustand z_1 und Zustand z_2 liegt, modelliert den Umstand, dass eine Operation p bei einer Instanz o von T genutzt werden kann, wenn sich o im Zustand z_1 befindet. Nachdem die Operation aufgerufen wurde, wechselt o in den Zustand z_2. Der temporale Aspekt des Instanzverhaltens wird somit durch Paare *(Zustand, Operation)* beschrieben (vgl. a. Kapitel 3.

Eine wesentliche Eigenschaft dieses Ansatzes ist die beschränkte Rolle der temporalen Aspekte als Beschreibung einer Folge von Operationen und Zustandsübergängen von Ausprägungen bestimmten Typs. Eine aktive Spezifikation der *Art und Weise*, in der eine Instanz Operationen ausführt, liegt außerhalb des konzeptionellen Rahmens der Methode, wird aber auch nicht benötigt, da im Komponentenmodell dieser Arbeit eine strikten Black-Box-Sicht gefordert wird, die den Blick auf die Funktion*weise* verwehrt (s.a. [Griffel, 1998]).

Sobald nicht nur Reihenfolgen, sondern auch nebenläufige, auf absolute Zeitpunktebezogene oder einen Historienverlauf betreffende Aktivitäten einer Komponente spezifiziert werden sollen, ist ein gegenüber dem EA potenziell besser geeigneter Formalismus durch die *Temporale Logik* gegeben. Ein solcher Ansatz ist beispielsweise im Rahmen von *Echtzeit*-Anforderungen an eine Applikation sinnvoll. Fragestellungen der Real-Time-Programmierung liegen zwar ausserhalb der Betrachtungen dieser Arbeit, doch sollen entsprechende Formalismen bereits im Rahmen des Komponentenmodells als vorbereitende Arbeiten untersucht werden, um eine spätere Ausweitung der Anwendbarkeit des Gesamtkonzepts zu erlauben.

Temporale Logik *Temporale Logik* ist eine Erweiterung der *modalen Logik*, die Aussagen der Form »möglicherweise«, »unter Umständen«, etc. erlaubt. Letztere werden um qualitative zeitliche Aspekte, die Beziehungen wie »ab jetzt«, »später einmal«, »als nächstes« oder »solange bis« auszudrücken vermögen.

Temporale Aussagenlogik hat in Bezug auf die Beschreibung zeitlicher Aspekte von Komponenten eine Reihe vorteilhafter Eigenschaften [Arapis, 1995]. Im Gegensatz zu Ansätzen mit endlichen Automaten ist es möglich, die temporalen Eigenschaften eines Objektes unmittelbar zu beschreiben, wobei Sequen-

zen funktionaler Einheiten beschrieben werden, die gesendet und empfangen werden. Es besteht bei diesem Ansatz nicht der Zwang, Zustände anzugeben, deren einziger Zweck darin besteht, dem Formalismus zu genügen. Trotzdem besteht die Möglichkeit, nach freier Entscheidung sinnvolle Zustände zu definieren, die dem besseren Verständnis zuträglich sind und die Verhaltensbeschreibung erleichtern.

Bezüglich ihrer grundsätzlichen Eigenschaften lässt sich abschließend feststellen, dass die im Komponentenkontext relevante Form der temporalen Aussagenlogik grundsätzlich *entscheidbar* ist. D.h. semantische Aussagen in Form von logischen Formeln lassen sich immer verifizieren. Zu diesem Zweck existieren effiziente Algorithmen zur Überprüfung der Erfüllbarkeit entsprechender Formeln, die in der Literatur umfassend betrachtet wurden und auf die hier daher nicht detailliert eingegangen werden soll [Gabbay, 1992] [Gabbay et al., 1992].

Typisierung im Komponentenkontext Typisierung ist im Bereich der Programmiersprachen ein weit verbreiteter Ansatz. Es stellt sich die Frage, ob diese auch im Rahmen von Komponentenarchitekturen eine nutzbringende Ergänzung darstellen würde, und welche Rolle ihr hierbei zugedacht werden soll.

Nützlich ist zunächst einmal die Tatsache, dass Typen *Invarianten* ihrer Instanzen darstellen. Daraus resultiert eine einfache Verifizierbarkeit korrekter Beziehungen zwischen Bausteinen anhand ihrer Typen, was allgemein als Typsicherheit bezeichnet wird. Gerade im Zusammenhang mit Komponenten, die mit ihrer Bestimmung zur Kompositionen ein ständiger Gegenstand wechselseitiger und potenziell wechselhafter Beziehungen sind, erscheint die Möglichkeit eines entsprechenden automatischen *Typprüfung* äußerst nützlich.

Durch Typisierung fließen zusätzliche (Typ-) Informationen in ein System ein. Diese Informationen erlauben es, bei der Auswertung – im Allgemeinen einer Kompilierung – *Optimierungen* vorzunehmen, die ohne entsprechende Informaitonen nicht möglich wären (vgl. a. Kapitel 5 und Abschnitt 6.3).

Der Komponentensicht entspricht der Wunsch, dass entsprechende Typinformationen beim Übergang der Entwicklungphase in den laufenden Betriebszustand *erhalten* bleiben, um beispielsweise im Falle einer evolutionären Systempflege ausgewertet werden zu können (vgl. a. [Benthien, 1999], [Martens, 1999], [Moehl, 1999b]).

Typsysteme unterstützen auch in direkter Hinsicht den Kompositionsgedanken. Da Typen ihrerseits in verschiedenartigen

Beziehungen stehen, und solche einen direkten Bezug zu denen der entsprechenden Instanzen aufweisen, lassen sich nützliche Rückschlüsse ziehen. Die Einführung von *Subtypbeziehungen* und *Typhierarchien* eröffnet hier die Möglichkeit einer *Typsubstitution*, die eine größere Flexibilität bei der Komposition von Applikationen erlaubt. Die Unterscheidung von *Typkonformität* und *Typkompatibilität*[46] erlaubt hierbei Rückschlüsse auf die Kompositionspartner. Eine Unterscheidung zwischen *Verhaltenstypen* und *Strukturtypen* kann sinnvoll sein. Abschnitt 4.1.1.2 geht hierauf nochmals ein.

2.3.4 Entwurf eines Komponentenmodells

Da Komposition der Schlüssel eines generativen Ansatzes ist [Batory et al., 1994] liegt das Augenmerk der weiteren Betrachtungen insbesondere auf den für eine Komposition besonders relevanten Schnittstellenbegriff, und zwar möglichst solchen Schnittstellen, die sich als (innere) Systemgrenzen in einer Domäne (vgl. Abschnitt 5.3) wiedererkennen lassen, um zu einer »natürlichen« (fachbezogenen) Strukturierung des Gesamtsystems zu gelangen.

Für die Strukturierung der einzelnen Komponenten des Systems sollte die Unterscheidung zwischen inneren und äußeren Komponentenschnittstellen eigentlich auf Grund des Bausteinansatzes[47] entfallen. Da sie aber andererseits für technische Optimierungen hilfreich sein kann, soll auf diese Unterscheidung hier nicht verzichtet werden.[48]

Das Modell ist *streng* hierarchisch, es werden also ausschließlich Abhängigkeiten zwischen jeweils einer Komponente und ihrer Umgebung (direkt angrenzenden Komponenten) zugelassen, jedoch nicht Abhängigkeiten durch Indirektionen über mehrere Ebenen in der Gesamtstruktur. Abhängigkeiten sollten in Form von möglichst einseitig linearen Referenzierungen erfolgen (vgl. a. Abschnitt 5.5.2), wobei die Referenzen ausschließlich Typinformationen einbeziehen, also nicht auf individuelle Instanzen

[46] Letztere auch als *Typinteroperabilität* bezeichnet. Zu den genauen Begriffsdefinitionen siehe Kapitel 3 und 4

[47] Ein physischer Baustein besitzt auf Grund seiner geschlossenen Dinglichkeit nur *äußere* Schnittstellen zur Komposition.

[48] Vgl. etwa die Kommunikation von Enterprise JavaBean-Komponenten untereinander über remote(!)-Schnittstellen [Sun, 2000], die eigentlich für den Zugriff durch den Klienten gedacht sind. Die Firma Sun arbeitet derzeit an der Bereinigung solcher Entwurfsschwächen.

verweisen sollten. Insbesondere sollte die *Parametrisierung* einer Komponente mit einer anderen ausgeschlossen werden, um eine zu enge Kopplung und das Verschwimmen der Komponentengrezen zu vermeiden.

Das Modell sieht vier Ausprägungen des Komponentenbegriffs vor: *Atomare* (primitive), *aggregierte* (engl. *composite*), *Ressourcen*komponenten (engl. *proxies*) sowie *Adapter*komponenten (Konnektoren) vor. Atomare Komponenten bilden dabei die funktionalen Basisbausteine ohne innere Struktur (black-box). Aggregierte Komponenten sind dann Kompositionen solcher atomaren Bausteine mit einer entsprechenden Betrachtung innerer Systemgrenzen. Mit dem Begriff der Ressourcenkomponente ist ebenso wie mit dem der Adapterkomponente keine Modellerweiterung verbunden, sondern diese Begriffe verweisen nur auf die jeweils besondere Rolle der entsprechenden Komponenten. Ressourcenkomponenten sind atomare Komponenten, die eine für das System notwendig Ressource, z.B. eine Datenbank, kapseln, um eine Komponentensicht auf diese zu erhalten. Adapterkomponenten sind Komponenten ohne eigentliche Anwendungsfunktionalität, die lediglich einer anpassenden Verbindung anderer (funktionaler) Komponenten miteinander dienen. Solche *Konnektoren* müssen *unabhängig* von der Implementationsweise der durch sie verknüpften Komponenten sein, d.h. also ihrerseits nur mehr Schnittstellen bzw. Typinformationen berücksichtigen, im Idealfall also nur von semantischen Gegebenheiten einer Domäne abhängen (vgl. a. Kapitel 3 und 4).

Die bisherige Diskussion hat zu berücksichtigende Faktoren und Ansätze für den Entwurf eines *Komponentenmodells* dargelegt. Die Darstellung ist dabei bewusst sehr allgemein gehalten, da sie als Ausgangspunkt für die in den nachfolgenden Abschnitten behandelten Beschreibungsmethoden dienen soll ohne zu frühe Einschränkungen zu implizieren. Diese Beschreibungsmethoden unterteilen sich im Folgenden in drei Abschnitte. Zunächst wird die Beschreibung von *Verhaltensaspekten* durch zeitlich / kausale Bedingungen mittels *temporaler Logik* behandelt. Im Anschluß wird eine *dynamische Komponentenspezifikation* mit Hilfe der *Feature-Logik* betrachtet, bevor eine abschließende Diskussion einer *Integration* dieser Ansätze erfolgt.

2.3.4.1 Realisierung der Modellebene

Wie schon in vorangegangenen Abschnitten geschehen, wird für die Darstellung des Beispielmodells eine Gliederung in Struktur und Verhalten vorgenommen. Die Betrachtung einer expli-

ziten Metaebene entfällt hier, da sie im Rahmen des Entwurfs des Klassifikationskonzeptes (vgl. Kapitel 4) betrachtet wird.

Strukturelle Modellbestandteile Der erste zentrale Begriff des Komponentenmodells ist die Komponente selbst, deren Eigenschaften sich an der bereits in Abschnitt 1.1.6 vorgeschlagenen Charakterisierungen orientieren. Die konkrete Begriffsbestimmung bedarf allerdings noch der Verfeinerung der für sie verwendeten Begriffe in den nachfolgenden Kapiteln (vgl. insbesondere Abschnitt 4.3 und 5.3):

Definition 2.2 (Komponente) *Eine* Komponente *ist die softwaretechnische Abstraktion eines domänenspezifischen Fachkonzepts, das mindestens eine semantisch geschlossene* Interaktion *im Sinne des vollständigen Austausches zugehöriger* Domänenaktivitäten *mit einer anderen Komponente ermöglicht.*

Diese Charakterisierung des Komponentenbegriffs betont zunächst dessen *problembezogene* – nicht etwa technisch motivierte – Herleitung aus dem fachspezifischen Einsatzumfeld eines Softwarebausteins. Auf diese Aspekte soll jedoch erst später eingegangen und im Weiteren zunächst funktionale und strukturelle Betrachtungen erörtert werden.

Zentraler Bestandteil einer Komponente ist der Komponentenkern, der deren konkrete Realisierung und dynamische Zustände kapselt. In ihm manifestieren sich ferner die anpassbaren Komponenteneigenschaften sowie – im Falle aggregierter Komponenten – die internen Abhängigkeiten der verschiedenen über Teilkomponenten eingebrachten Komponentenschnittstellen.

Da eine logische Untergliederung der Komponentenfunktionalität eine innere Abhängigkeit der Teilfunktionen impliziert(!) kann im Falle von Mehfachschnittstellen nicht auf interne Spezifikationen verzichtet werden. Das Modell dieser Arbeit sieht jedoch vor, dass Mehrfachscnhnittstellen ausschließlich bei aggregierenden Komponenten auftreten, nicht aber bei atomaren.

Komponenten stellen ihre Funktionalität nach außen über Mengen eindeutiger, klar abgegrenzter Exportschnittstellen bereit, deren Spezifikation potenziellen Nutzerkomponenten (Kopplungspartnern) im Sinne eines Vertragsverhältnisses Zusicherungen garantiert. Die Anforderungen einer aggregierenden – nicht atomaren – Komponente an ihre potenziellen Subsysteme wird durch vorausgesetzte Importschnittstellen ausgedrückt.

Der obige Komponentenbegriff enthält als Bestandteile *Komponentenkern* und *-schnittstellen*. Diese Konzepte werden nachfolgend beschrieben.

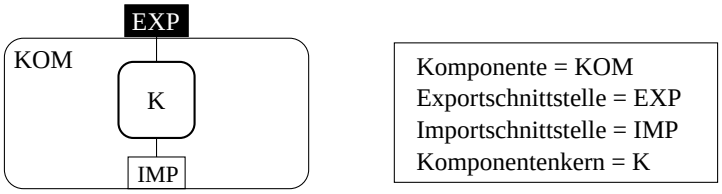


Abbildung 2.11
Atomare Komponente

Begriff 1 (Komponentenkern) *Ein* Komponentenkern *(vgl. Abbildung 2.12) ist der zentrale Bestandteil einer atomaren Komponente und dient als gemeinsamer* ***konzeptueller, interner Bezugspunkt****.*[49] *Komponentenintern ist er Bindeglied zu den Aspekten von Implementation und Zustand. Komponentenextern ist er Träger der Abstraktionen von Komponenteneigenschaften sowie einer zentralen Komponenten-übergreifenden Spezifikation.*

Letztere liegt in Form formaler Beschreibungen der bereitgestellten und vorausgesetzten Funktionalität der Komponente vor, welche sowohl statische als auch dynamische Anteile umfassen, wobei letztere in Form temporaler bzw. operationaler Zusammenhänge berücksichtigt werden.

Die Komponentenspezifikation besteht im einzelnen aus folgenden Teilen:

- *Eine* ***Menge von Spezifikationen*** *der Komponentenschnittstellen, aufgeteilt in Teilmengen bezüglich der Import- und Exportteile,*

- *Einem* ***zentralen temporalen Aspekt*** *der wechselseitigen Abhängigkeiten aller durch die Schnittstellenmenge gegebenen importierten und exportierten Funktionalitäten sowie der internen Zustände.*

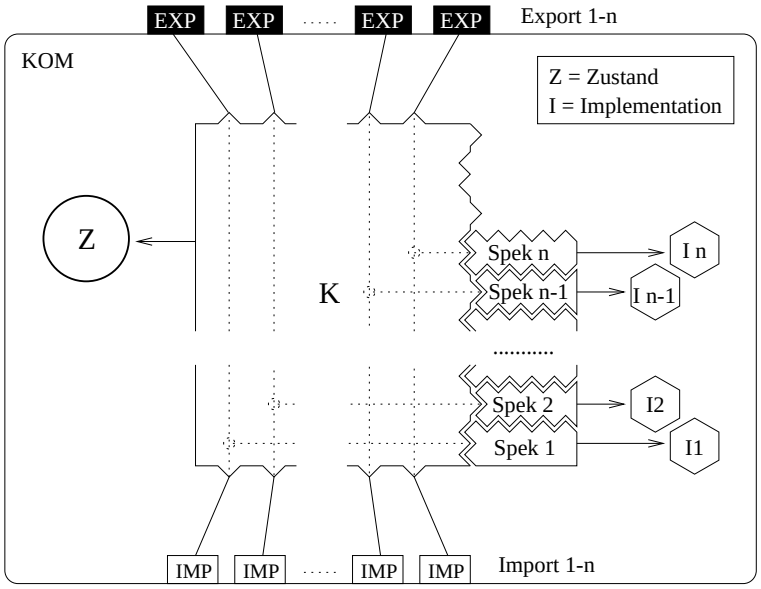
Um eine Erweiterbarkeit der Komponente zu ermöglichen, lässt sich der Kern um Spezifikationsbestandteile ergänzen.

Einen Komponentenkern zu postulieren ergibt nicht nur im Falle atomarer Komponenten Sinn. Aggregierte Komponenten besitzen zwar eine inhärente Modularisierung, doch spiegelt der Begriff des Kerns in diesem Fall die Menge der entsprechenden Ein-

[49] Ein solcher »Fixpunkt« ist insbesondere für einen möglichen Rollenwechsel einer Komponente (vgl. a. Abschnitt 4.8) sowie in Form einer konkreten technischen Ausprägung für die evolutionäre Modifikation bzw. Aktualisierung [Griffel, 1998] einer Komponente von Bedeutung.

zelspezifikationen wider, allerdings unter Beibehaltung deren gegenseitiger Unabhängigkeit (segmentiertes Modell).[50]

Abbildung 2.12
Der Komponentenkern



Begriff 2 (Komponentenschnittstelle) Schnittstellen *fungieren als explizite* ***externe Bezugspunkte*** *für Komponentenbeziehungen. Sie dienen dabei der eindeutigen Spezifikation von Funktionalität in sowohl statischem als auch dynamischem und damit semantisch reichem Sinne und stellen einen Vertrag zwischen nutzender und erbringender Komponente dar. Schnittstellen teilen sich in Importschnittstellen und Exportschnittstellen auf, welche die Bereiche vorausgesetzter und bereitgestellter Funktionalität abdecken. Aus beiden Bereichen können mehrere Schnittstellen pro Komponente vorliegen. Atomare Komponenten besitzen ausschließlich Exportschnittstellen. Aggregierende Komponenten zeichnen sich durch zusätzliche Importschnittstellen aus.*

Eine ***Exportschnittstelle*** *besteht aus folgenden Teilen:*

- Statische Strukturinformationen *der exportierten Funktionalität, d.h.* ***Namen*** *von Schnittstelle, Nachrichten als verwendeten syntaktischen Einheiten und Attributen zur Modellierung von Zuständen.*

[50] Das Typmodell des Kapitels 4 fokussiert im Sinne eines homogenen geschlossenen Komponentenmodells auf solche Aggregationen.

- Semantische Voraussetzungen *über die Anwendung der exportierten Funktionalität. Diese liegen in Form des* ***temporalen Aspektes*** *über den Nachrichten und Attributen einer Schnittstelle vor, welcher deren zeitliche Abhängigkeiten ausdrückt. Der temporale Aspekt einer Schnittstelle leitet sich aus dem zentralen* Komponentenkern *ab.*

Eine ***Importschnittstelle*** *besteht aus folgenden Teilen:*

- Statische Strukturinformationen *über importierte Funktionalität in Form von vorausgesetzten* Komponenten, Schnittstellen *oder* Nachrichten *und* Attributen.
- Semantische Anforderungen *bezüglich der Verwendung der importierten Funktionalität. Diese liegen in Form des temporalen Aspektes über der Menge von Nachrichten bzw. Attributen vor. Der temporale Aspekt leitet sich wiederum aus dem zentralen* Komponentenkern *ab.*

Wird die Betrachtung einer Komponentenstruktur auf deren Schnittstellen eingeengt, so ergibt sich eine korrespondierende Schnittstellenstruktur mit wechselseitigen Anforderungen und Bedingungen bezüglich statischer und dynamischer Eigenschaften der Gegenstücke. Abbildung 2.13 veranschaulicht diese Sicht für eine beispielhafte Konfiguration mit vier Import- und Exportschnittstellen.

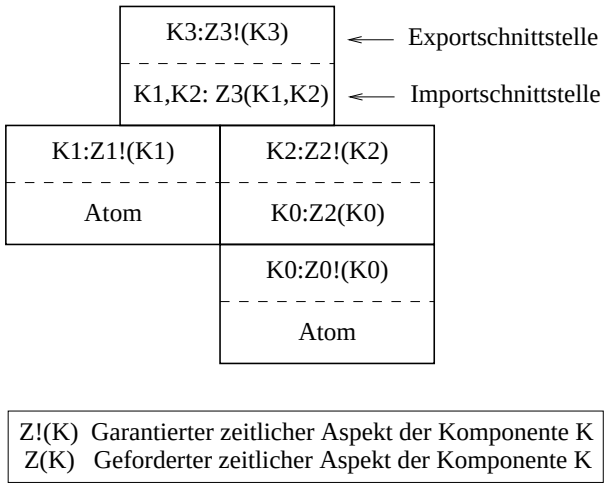


Abbildung 2.13 *Beispiel einer Schnittstellenstruktur*

Wie aus dem Schnittstellenbegriff hervorgeht, sollen als atomare Interaktionseinheiten im Rahmen dieses Beispielmodells *Nachrichten* dienen:

Begriff 3 (Nachricht) *Eine* Nachricht *ist die Abstraktion einer* ***atomaren Interaktionseinheit***[51] *zwischen* Komponenten *mit eindeutiger, expliziter Semantik. Diese Semantik wird innerhalb des Modells als durch die Analyse der Einsatzdomäne gegeben betrachtet und nicht weiter aufgeschlüsselt. Nachrichten werden zwischen den* Schnittstellen *genau einer sendenden und einer empfangenen* Komponente *ausgetauscht. Nachrichten werden in diesem Sinne bezüglich der betrachteten Schnittstelle immer als entweder eingehend oder ausgehend gekennzeichnet.*

Um die Möglichkeit zu schaffen, relevante Zustände modellieren zu können, werden als letzte strukturelle Modellbestandteile *Komponentenattribute* eingeführt:

Begriff 4 (Attribut) *Ein* Attribut[52] *ist die Abstraktion einer* ***Komponenteneigenschaft*** *mit eindeutiger, expliziter Semantik, welche wiederum als durch die Einsatzdomäne gegeben betrachtet wird. Das Lesen bzw. Setzen eines Attributes erfolgt durch spezielle* Nachrichten*. Da Nachrichten nicht parametrisiert sind, sei für jedes Element des Attributwertebereichs eine korrespondierende Nachricht gegeben.*

Der Komponentenbegriff beinhaltet deren Intention zur zielgerichteten wechselseitigen Kooperation. Einzelne Zusammenschlüsse dieser Art werden als *Kompositionen* bezeichnet und wie folgt festgelegt:

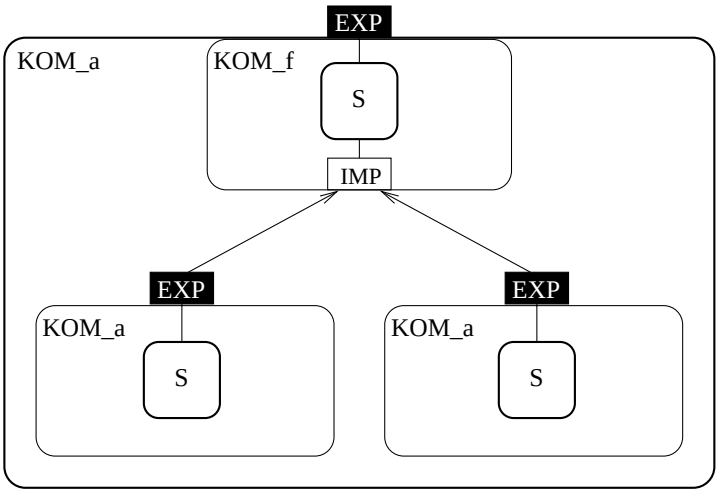
Begriff 5 (Komponentenkomposition) *Eine* Komponentenkomposition *ist die Kombination von Komponenten zu einem Aggregat, das wiederum eine Komponente darstellt (Abb. 2.14). Die konstituierenden Komponenten werden dabei als Teilkomponenten bezeichnet, deren Konglomerat auch als Aggregat, Verbund- oder Toplevelkomponente.*

Als Bedingung *für die Komposition müssen die* vertraglichen Bedingungen *aller Kooperationspartner, welche in deren* Komponentenschnittstellen *bzw.* Komponentenkernen *spezifiziert sind, eingehalten werden. Dies kann dabei auch indirekt unter Zuhilfenahme von Adaptions- und Interoperabilitätmechanismen zur Integration heterogener Komponenten erfüllt werden.*

[51] *Domänenaktivitäten* (vgl. Kapitel 4) setzen sich aus solchen Nachrichten zusammen.

[52] Attribut ist hier nicht als Merkmal im Sinne von Abschnitt 5.3 aufzufassen, sondern als Eigenschaft im Sinne eines (programmiersprachlichen) »Property«.

Als Beschränkung *sei festgelegt, daß Kompositionen nicht zu unmittel- oder mittelbaren* Zyklen *innerhalb einer Komponentenstruktur führen dürfen. Ferner sei eine* multiple Verwendung *– also die Kombination einer Komponente als Bestandteile mehrerer verschiedener Toplevelkomponenten – ausgeschlossen.*



Komponente atomar = KOM_a
Komponente im Framework = KOM_f

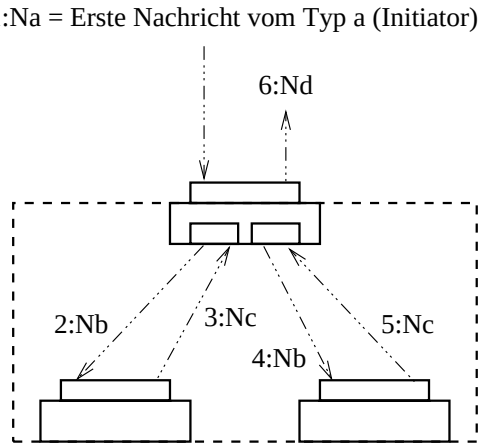
Abbildung 2.14 *Komponentenkomposition aus Modellsicht*

Modellverhalten Die grundlegende Aktivität innerhalb des vorliegenden Komponentenmodells besteht im wechselseitigen *Nachrichtenaustausch*. Ein solcher Nachrichtenaustausch findet zwischen einer sendenden und einer empfangenen Komponente statt. Der Zeitpunkt des Austausches ergibt sich aus der Bereitschaft beider Kommunikationspartner. Die Menge möglicher Nachrichten ist auf eine endliche Anzahl *beschränkt* gemäß der in der Aufbereitung der zugehörigen Domäne gefundenen Interaktionen.

Die Bausteine des Systems sollen grundsätzlich unabhängig agieren, d.h. das System stellt eine *nebenläufige* Umgebung dar. Die Vermeidung typischer Probleme wie einer Verklemmung diskutiert Kapitel 3. Die dort diskutierten adaptierenden Konnektoren verwalten einen Zustand pro Verbindung zweier Kopplungspartner, sodass im Falle nebenläufiger Nutzung einer Komponente, diese nicht selbst unterscheiden muss, mit welcher anderen Komponente sie gerade in Interaktion tritt).

Aus den atomaren Aktivitäten des Nachrichtenaustausches setzen sich wiederum komplexere *Interaktionen* zwischen Komponenten zusammen, die als *Domänenaktivitäten* bezeichnet werden sollen (vgl. Kapitel 4). Interaktionen finden zu einem gegeben Zeitpunkt der Betrachtung immer zwischen genau zwei Komponenten statt, wobei eine der beiden immer die Toplevelkomponente einer Komposition ist. Diese Toplevelkomponente ist stets *Initiator* der Aktivität. Zwei Teilkomponenten einer Struktur können in diesem Sinne immer nur indirekt über die Toplevelkomponente interagieren, welche in diesem Fall die Funktion eines *Mediators* übernimmt. Abbildung 2.15 visualisiert diese Zusammenhänge (vgl. a. Kapitel 3).

Abbildung 2.15
Komponenteninteraktion als Nachrichtensequenz



Die erlaubten globalen Nachrichtensequenzen ergeben sich aus den, bezüglich der beteiligten Komponenten, lokalen Gegebenheiten. Diese Einzelkomponenten garantieren auf der einen Seite ihr eigenes Verhalten in Form von Sequenzen über einer Menge eigner Nachrichten und fordern auf der anderen Seite ein ebenso bestimmtes Verhalten ihrer Partner. Um solche lokalen Bedingungen explizit zu machen, dienen formale Beschreibungen der temporalen Aspekte, welche über Schnittstellen offengelegt bzw. über Konnektoren aktiv berücksichtigt werden (vgl. Kpaitel 3). Abbildung 2.16 veranschaulicht das sich ergebende Netzwerk wechselseitiger Beziehungen.

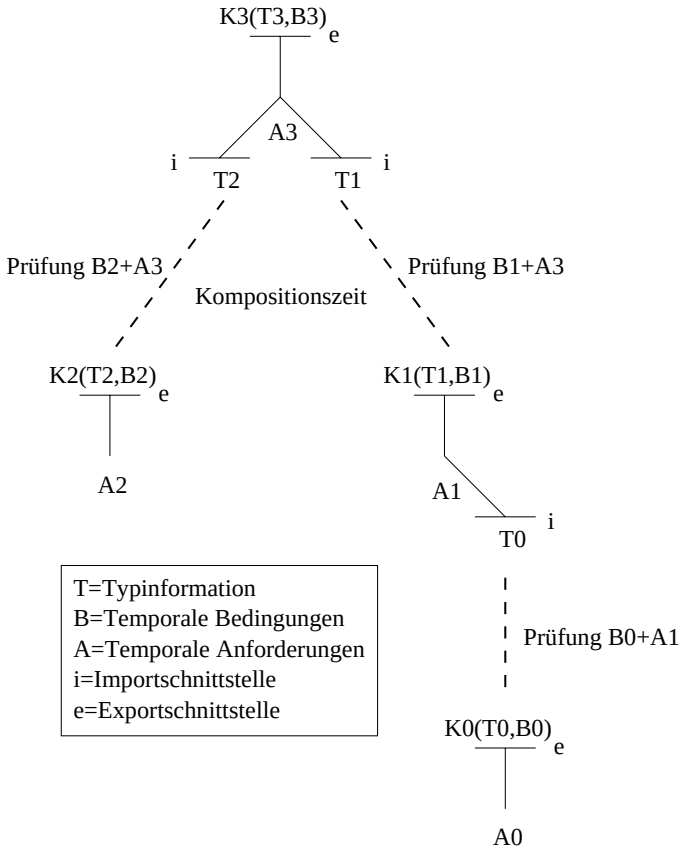


Abbildung 2.16
Globale Interaktion als Wechselwirkung lokaler Verhaltensaspekte

2.3.5 Verhaltensbeschreibung mittels Temporallogik

Temporallogik Temporale Logiken existieren in zahlreichen unterschiedlichen Ausprägungen (vgl. Abbildung 2.17). Zu unterscheiden ist zunächst nach Art der zugrundeliegenden klassischen Logik zwischen *propositionaler*- bzw. *Aussagenlogik* sowie *Prädikatenlogik*. Ferner existieren bezüglich der Modellierung der Zeit *diskrete* und *kontinuierliche* Varianten. Bezüglich des so modellierten Zeitbegriffs, findet eine weitere Differenzierung nach dessen Eigenschaften statt. Aus den vielfältigen Ausprägungen temporaler Logik muss die Menge solcher Varianten herausgefiltert werden, die zur Beschreibung des temporalen Aspektes von Komponenten geeignet erscheint.

Zunächst sollte eine geeignete *Grundform der Logik* bestimmt werden. Formal betrachtet ergeben sich dabei zwei we-

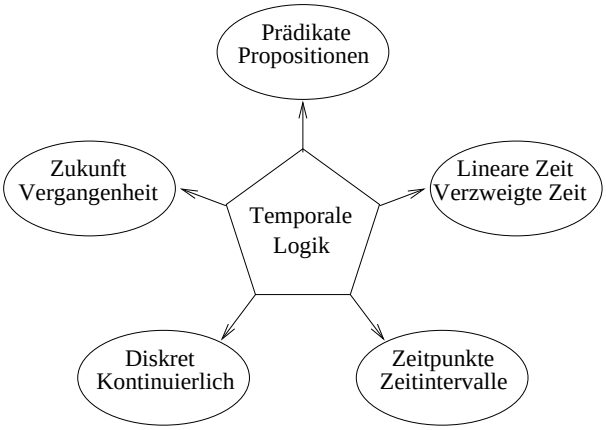


Abbildung 2.17 *Formen temporaler Logik*

sentliche Fakten. Zum einen ist Prädikatenlogik in Bezug auf die Aussagekraft mächtiger als propositionale Logik, zum anderen ist aber Prädikatenlogik im Gegensatz zu propositionaler Logik nicht generell entscheidbar [Arapis, 1991]. Während für die propositionale Variante effiziente Algorithmen zum Test der Erfüllbarkeit bestehen, die eine automatisierte Verifikation entsprechender Spezifikationen ermöglichen, ist dies für eine entsprechende Prädikatenlogik nicht der Fall. Liegt der Fokus auf der praktischen Anwendbarkeit, sollte somit von letzterer abgesehen werden. Darüber hinaus hat sich die propositionale Logik in vielen Untersuchungen als geeignet erwiesen.[53].

Weiter sollte eine sinnvolle Form des *Zeitbegriffs* gewählt werden. Temporale Logiken existieren für verschiedene *Zeitlinien*. Diese können ihren Anfang in der unendlichen *Vergangenheit* oder einem konkreten Punkt – namentlich der Gegenwart – haben und von dort in die *Zukunft* verlaufen. Ferner lässt sich zwischen *eindeutigen* und *verzweigten* Zeitlinien unterscheiden. Es stellt sich die Frage, welches Zeitmodell im Rahmen des Komponentenparadigmas am sinnvollsten ist.

Wird alleine die *initiale Komposition* einer komponentenbasierten Anwendung betrachtet, so sind eindeutige Zeitlinien, welche vom Punkt der Gegenwart in die Zukunft verlaufen, geeignete Ausdrucksmittel, denn die entsprechenden Systeme sind vorher noch nicht existent. Ihr Dasein beginnt im Kompositionszeitpunkt und verläuft von dort an eindeutig in die Zukunft. Ein Ver-

[53] Sie wurde insbesondere mehrfach erfolgreich im Zusammenhang mit Prozeßbeschreibungen angewandt [Manna & Wolper, 1984]

gangenheitsbezug auf die Zeit vor der Initialisierung ist somit nicht relevant.

Wird hingegen die *Rekonfiguration* oder *Evolution* bestehender Systeme mit langlebigen, wiederverwendbaren Komponenten betrachtet, kann die Antwort nicht so pauschal ausfallen. Hier wird die *Historie* relevant. Zur Behandlung dieses Szenarios sind zwei Ansätze denkbar.

Zunächst wäre es in dieser Situation denkbar, den Startpunkt einer Zeitlinie – also die Gegenwart des temporalen Systems – in die reale Vergangenheit zu legen. Dies ist möglich, da die relevante Historie in jedem Fall nur endlich weit in die Vergangenheit reicht, nämlich maximal bis zum Kompositionszeitpunkt des betrachteten Systems. Somit kann die gesamte relevante Zeitspanne allein mit Hilfe der eingeschränkten, zukunftsorientierten Sicht abgedeckt werden. Nachteil dieses Vorgehens ist die Verfälschung der realen Zeitverhältnisse. Soll dies vermieden werden, soll also die Modellgegenwart mit der formalen Gegenwart übereinstimmen, muss auf eine Zeitlinie mit Vergangenheit und Zukunft zurückgegriffen werden, die zu komplexeren Methoden führt. Aus rein formaler Sicht ist dies jedoch nicht notwendig, denn die theoretische Ausdruckskraft erhöht sich dadurch nicht [Gabbay, 1992]. Dies soll auch hier Argument für die Wahl rein *zukunftsorientierter Zeitlinien* im vorliegenden Modell dienen.

Die wesentlichen Ereignisse des interaktionsorientierten Komponentenmodells umfassen das Senden und Empfangen von Nachrichten von deren Dauer in dieser Arbeit abstrahiert werden soll.[54] Entsprechend ist die Wahl von *diskreten Zeitpunkten* sinnvoll ausreichend.

Die Verwendung verzweigender statt linearer Zeitlinien kann hier allerdings nicht generell ausgeschlossen werden. Eine entsprechende Untersuchung sprengt jedoch den Rahmen dieser Arbeit. Als Fazit resultiert nun ein Zeitbegriff mit einer unendlichen Anzahl diskreter, zukunftsgerichteter Ereigniszeitpunkte. Die entsprechende Zeitlinie ist in Abbildung 2.18 illustriert.

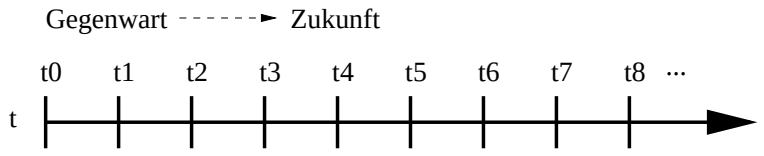


Abbildung 2.18
Lineare Zeitlinie mit diskreten Zeitpunkten

[54] Ein entsprechendes Realzeit-Verhalten einer Software wird wie angesprochen hier nicht berücksichtigt.

Aus klassischer Grundlogik und Zeitcharakteristik ergibt sich der im weiteren verwendete Formalismus, nämlich die *propositionale lineare temporale Logik* (*PLTL*). Die formale Einführung dieser Logik findet sich in Anhang H, in dem auch die im weiteren Verlauf der Arbeit diesbezüglich verwendete Notation und Terminologie angegeben ist.

2.3.5.1 PLTL basierte Formalisierungskonzepte des Komponentenmodells

Für die Beschreibung zeitlicher Aspekte von Komponenten mittels temporaler Logik gibt es verschiedene denkbare Varianten. Zunächst wird eine Vorgehen betrachtet, das auf zentralen logischen Ausdrücken beruht. Als mögliche Erweiterung wird dann eine verfeinerte, untergliederte Spezifikationsform diskutiert. Schließlich wird die Rolle von Propositionen untersucht und deren Ausdrucksfähigkeit kritisch hinterfragt.

Grundform der Spezifikation Der erste mögliche Ansatz besteht in einer Beschreibung des Komponentenverhaltens durch zentrale, d.h. im Falle eines Aggregats schnittstellenübergreifender temporallogischer Ausdrücke. Das Modell lehnt sich an den entsprechenden Ansatz von Arapis an [Arapis, 1995].

Strukturelle Bestandteile Als strukturelle Bestandteile der Spezifikation werden ausschließlich Nachrichten betrachtet. Schnittstellen hingegen werden durch Nachrichtenmengen abgebildet (vgl. a. Kapitel 4. Unterschieden werden dabei eine Exportsowie mehrere Importschnittstellen und entsprechend eine Menge *öffentlicher* und mehrere Mengen *subkomponentenspezifischer* Nachrichten. *Attribute* werden bei diesem Ansatz durch deren korrespondierende Nachrichten berücksichtigt.

Temporale Aussagen Jede Nachricht einer Komponente wird mit einer entsprechenden atomaren Proposition in PLTL assoziiert. Die Tatsache, daß einer Proposition p in der zu einem Zeitpunkt (t) gehörenden Interpretation der Wert »wahr« zugeordnet wird, bedeutet, dass die mit p assoziierte eingehende (ausgehende) Nachricht in diesem Zeitpunkt von der betrachteten Komponente empfangen (gesendet) wird. Dabei wird vorausgesetzt, dass immer genau eine Nachricht zur Zeit pro Komponente gesendet oder empfangen wird.[55] Mit den Propositionen kann

[55] D.h. in jeder Interpretation wird genau einer Proposition der Wert »wahr« und allen anderen der Wert »falsch« zugeordnet. Eine entspre-

die zeitliche Abhängigkeit zwischen Nachrichten durch PLTL-Formeln ausgedrückt werden. Diese Formeln stellen grundlegende temporale Bedingungen einer Komponente dar. Sie werden in zwei Mengen eingeteilt, wobei die erste Menge öffentlicher Bedingungen dem Exportanteil entspricht und nur die entsprechenden öffentlichen Nachrichten berücksichtigt. Die zweite Menge teilkomponentenspezifischer Bedingungen kann alle Nachrichten verwenden und entspricht der zentralen Importspezifikation eines Aggregats.

Beispiel Zur Illustration soll das schon früher verwendete Beispiels einer Maschine mit Produktions- und Wartungszyklus beschrieben werden. Die entsprechende atomare Komponente besitzt ein Attribut, mit dem ihr Zustand modelliert wird, der entweder »Wartung« oder »Produktion« sein kann. Sie kann zudem zwei Nachrichten empfangen, welche den Start oder Stopp der Produktionsphase einleiten. Eine exemplarische Notation dieser Beschreibung ist in Abbildung 2.19 gezeigt.

Komponente: $Maschine;$
Attribute:
$Status: \{Produktion, Wartung\};$
Öffentliche Nachrichten:
$\downarrow Start, \downarrow Stop;$
Öffentliche Bedingungen:
$Status := (Produktion \vee Wartung);$
$\Box(Start \Rightarrow ((Status == Produktion) \wedge \triangle Stop));$
$\Box(Stop \Rightarrow (\blacktriangle Start \wedge \triangle((Status := Wartung) \vee Start)));$
$\Box((Status == Wartung) \Rightarrow (\triangle(Status := Produktion)));$

Abbildung 2.19 *Exemplarische Komponentenspezifikation »Maschine«*

Die Spezifikation enthält ebenfalls eine Menge temporale Bedingungen der Komponente. Attributspezifische Zuweisungsnachrichten werden in den Formeln durch die übliche Notation mit $:=$ abgekürzt. Die Abfrage eines Attributs entspricht einer Teilformel, welche die frühere Zuweisung des entsprechenden Wertes garantiert und wird durch $==$ abgekürzt. Die erste Bedingung besagt in diesem Sinne, dass der Zustand der Maschine am Anfang entweder »Produktion« oder »Wartung« ist. Die zweite Bedingung sagt aus, dass in allen zukünftigen Zeitpunkten eine »Start« Nachricht stets impliziert, dass der Zustand gleichzeitig »Produktion« ist und im nächsten Ereigniszeitpunkt

chende Auflösung nebenläufiger Aktivitäten geschieht dabei ausschließlich über das puffernde Verhalten der Konnektoren (vgl. Kapitel 3).

eine »Stop« Nachricht empfangen wird – also weder ein Zustandswechsel noch eine weitere »Start« Nachricht dazwischen liegen.

Die dritte Bedingung bezieht sich auf das Produktionsende. Sie enthält einen bisher nicht betrachteten temporalen Operator $\blacktriangle p$, der sich auf die Vergangenheit bezieht und besagt, dass p genau im vorherigen Zeitpunkt den Wert »wahr« hatte. Bedingung drei meint dann also, dass in allen zukünftigen Zeitpunkten eine »Stop« Nachricht stets impliziert, dass die unmittelbar vorhergehende Nachricht »Start« war und dass im direkt folgenden Zeitpunkt die Maschine entweder durch entsprechende Zuweisung den Status »Wartung« annimmt oder durch eine Nachricht »Start« die Produktion ohne Zwischenstop wieder aufnimmt. Um die Wartungsphase zeitlich zu begrenzen, drückt die letzte Bedingung aus, dass in allen zukünftigen Zeitpunkten der Status »Wartung« impliziert, dass im unmittelbar folgenden Ereigniszeitpunkt wieder der Zustand »Produktion« zugewiesen wird und der Betrieb somit fortgesetzt werden kann. Zu dieser Spezifikation gehört eine entsprechende Menge konformer Nachrichtensequenzen, die den temporalen Bedingungen genügt. Eine davon ist mit der zugehörigen Zeitlinie in Abb.2.20 dargestellt.

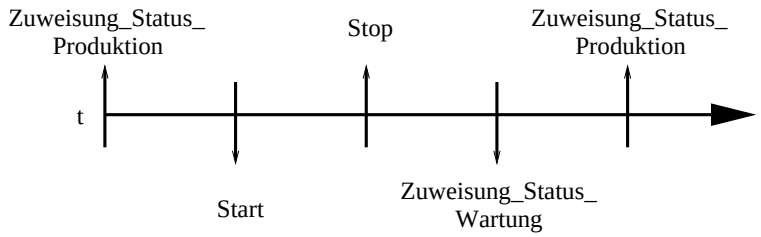


Abbildung 2.20 *Mögliche korrekte Nachrichtensequenz der Maschine*

Lokale und globale Zeit Ein wichtiger Aspekt temporaler Verhaltensbeschreibungen ist die Unterscheidung von *globaler* und *lokaler Zeit*. Eine PLTL-Formel bezieht sich in ihrer Grundform nur auf die komponenteninterne, lokale Zeitrechnung, d.h. Abfolge von Ereigniszeitpunkten. Bei der Beschreibung von Vorgängen zwischen Komponenten muss hingegen eine globale Zeitrechnung zugrundegelegt werden, die für alle beteiligten Komponenten gleichermaßen gilt. Auf Grund der einfacheren Spezifikation und größeren Übersichtlichkeit ist es jedoch wünschenswert, einzelne temporale Aussagen trotzdem lokal auszudrücken und erst später in Bezug zum Gesamtsystem zu bringen. Die entsprechende Transformation der lokalen in eine globale Zeit unter Bei-

behaltung temporaler Aussagen wird als *Universalisierung* (engl. *universalization*) bezeichnet.

Aggregate Ein *Koponentenverbund* (Aggregat) wird über die zugehörige Toplevelkomponente spezifiziert. Die teilkomponentenspezifischen Nachrichten und Bedingungen definieren dabei die extern geforderten strukturellen Einheiten und deren zeitliches Verhalten. Jede dieser Nachrichten ist dabei mit einem Bezeichner der zugehörigen Teilkomponente versehen und hat dort ein entsprechendes Gegenstück. Die Verwendung öffentlicher Nachrichten der Toplevelkomponente erlaubt die Berücksichtigung derer Exportbedingungen zur übergreifenden Synchronisation. Da bei jedem Nachrichtenaustausch die Toplevelkomponente entweder als Sender oder Empfänger auftritt, wird deren Modellsicht als Mediator hier sehr gut abgebildet.

Verifikation Der wichtigste Grund einer Komponentenspezifikation ist die sich daraus ergebende Möglichkeit einer Verifikation der in einer korrekten Komposition erwarteten Funktionalität. Das beschriebene Modell erlaubt in diesem Sinne eine zweistufige Vorgehensweise.

Zunächst muss die Konsistenz der Spezifikationen einzelner, atomarer Komponenten geprüft werden. Dies geschieht durch Konjunktion der öffentlichen temporalen (Export-) Bedingungen, die dann auf Erfüllbarkeit geprüft werden. Nach diesem Konsistenztest ist sichergestellt, dass die Komponenten für sich genommen ohne interne Konflikte innerhalb von Kompositionen angewandt werden können.

Als zweiter – und wesentlicher – Schritt wird die Konsistenz von Komponentenstrukturen überprüft. Dies geschieht durch sukzessive Verifikaton der einzelnen beteiligten Aggregate. Abbildung 2.21 zeigt eine solche Struktur, die sich aus zwei Aggregaten zusammensetzt.

Zum Test eines Aggregats ist es dabei immer nur notwendig, die unmittelbar importierten (Teil-) Komponenten zu berücksichtigen. Von deren potenziell aggregierten, innerer Struktur kann abstrahiert werden. In diesem Sinne erfolgt zum Konsistenztest eine Konjunktion von Importbedingungen der Toplevelkomponente und allen Exportbedingungen der verwendeten Teilkomponenten. Die entsprechende, vollständige Spezifikation der Beispielstruktur ist in Abb. 2.21 gezeigt.

Als zusätzliche Bedingung ist die interne Kompatibilität der Import- und Exportbedingungen von Toplevelkomponenten sicherzustellen, d.h. jede Nachrichtensequenz, die die Importbe-

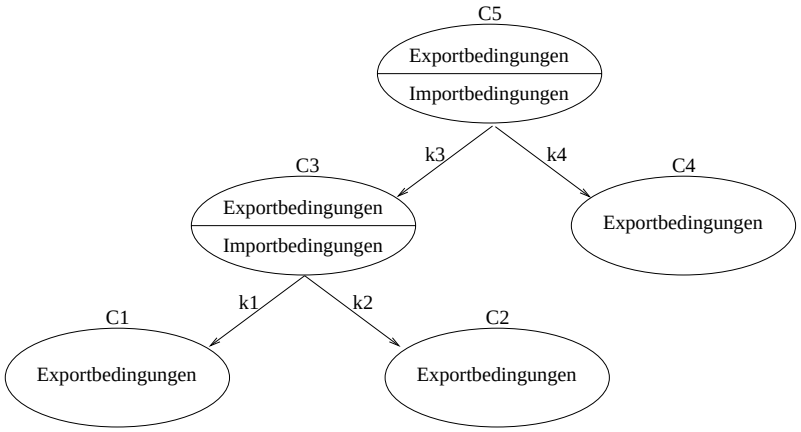


Abbildung 2.21
Verifikation einer Komponentenstruktur

$$
\begin{aligned}
Komposition\,C3 &= Importbedingungen - C3 \wedge \\
&\quad Exportbedingungen - C1 \wedge \\
&\quad Exportbedingungen - C2 \\
Komposition\,C5 &= Importbedingungen - C5 \wedge \\
&\quad Exportbedingungen - C3 \wedge \\
&\quad Exportbedingungen - C4
\end{aligned}
$$

dingungen erfüllt, muss bei Beschränkung auf die öffentlichen Nachrichten auch den Exportbedingungen genügen:

$$Importbedingungen \Rightarrow universalisierteExportbedingungen$$

aus. Die Universalisierung der Exportbedingungen ist notwendig, da zwischen den öffentlichen, prinzipiell beliebige aggregatinterne Nachrichten liegen können. Universalisierung ist in diesem Sinne ein notwendiger Wechsel von lokaler auf globale Zeit.

Fazit Das vorgestellte Konzept in Anlehnung an Arapis [Arapis, 1995] ist zunächst ein tragbarer Grundansatz für einen Beitrag zur formalen Spezifikation innerhalb des Komponentenmodells. Es ist vollständig und mathematisch beweisbar [Arapis, 1991, Arapis, 1992]. Zu bedenken ist allerdings, dass die Grundgedanken der ursprünglichen Arbeit erstens auf einer objektorientierten Sicht beruhen und zweitens auf eine Entwurfsmethodologie abzielen. Es ist daher angebracht, den Ansatz in Hinblick auf die in der vorliegenden Arbeit behandelte Thematik der Auswertung komponentenbasierter Systemarchitekturen zu überdenken und entsprechend zu optimieren.

Verfeinerung der Spezifikationsstruktur Ein Defizit der zuvor dargestellten Vorgehensweise ist deren stark zentralisierte und komprimierte Form. Einzelne Aspekte der Spezifikation werden dabei nicht separiert betrachtet. Dies kann zum einen dazu führen, dass verschiedene Aspekte in einzelnen Aussagen vermascht werden, zum anderen aber auch dazu, dass sich einzelne Aspekte zergliedern und über eine Vielzahl von Einzelaussagen verstreuen.[56] Als Folge werden die im Komponentenmodell hervorgehobenen multiplen Exportschnittstellen nicht getrennt betrachtet, und generell ist sowohl die Übersichtlichkeit vorhandener als auch die einfache Erstellung neuer Komponentenbeschreibungen fraglich. Im Folgenden werden unter Rückgriff auf das dem vorherigen Ansatz zugrundeliegende Modell einige Ansätze vorgeschlagen, die einer feineren, übersichtlicheren Gliederung der Spezifikation zuträglich sein könnten.

Adaption einer PLTL-Prozeßsynchronisation Die im vorangehenden Abschnitt betrachtete Möglichkeit formaler Spezifikation begründet sich auf einem Modell zur Synchronisation nebenläufiger Prozesse nach Manna und Wolper [Manna & Wolper, 1984]. Ausgehend von diesem Grundansatz sollen nun einige Anmerkungen zur PLTL-gestützten Spezifikation des Komponentenmodells folgen.

In dem ursprünglichen Modell von Manna / Wolper werden Mengen kommunizierender, sequentieller *Prozesse* betrachtet. Diese Prozesse, welche in Form von *CSP* (*Communicating Sequential Processes*) [Hoare, 1978] Programmen vorliegen, untergliedern sich jeweils in einen *funktionalen* und einen *synchronisierenden* Teil. Letzterer besteht dabei aus wechselseitigen, atomaren Operationsaufrufen und wird durch PLTL-Formeln beschrieben.

Die Beschreibungen synchronisierender Teile aller Prozesse des betrachteten Systems können kombiniert und auf Erfüllbarkeit geprüft werden. Aus den gefundenen Modellen wird ein zentraler, *Synchronisator* genannter Prozess abgeleitet[57], der den synchronisierenden Teil des Gesamtsystems darstellt und bei allen Interaktionen wirksam ist. Der Ansatz ist in Abbildung 2.22-A skizziert.

[56] Dies widerspräche dem Wunsch der Komponentensicht nach einer klaren Identifikation und Trennung der Belange (separation-of-concerns).

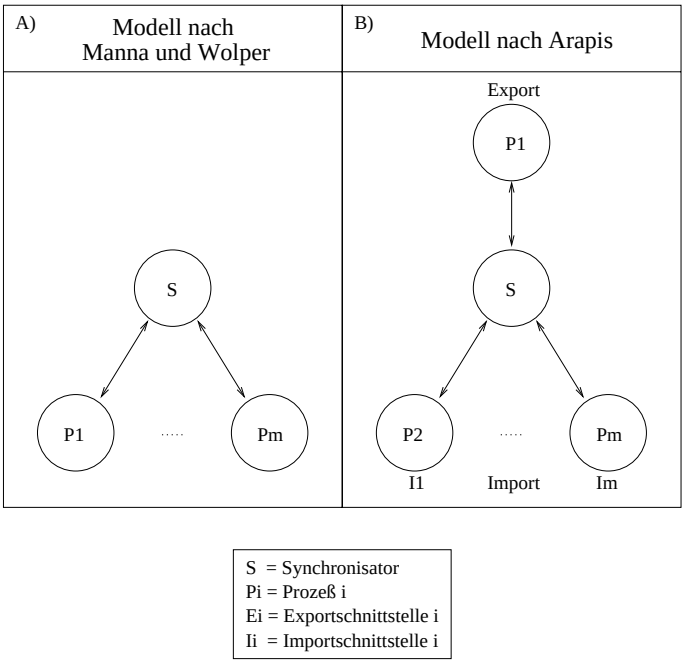
[57] Dieser ist in Analogie dem »Skelett« einer komponentenbasierten Anwendung zu zuordnen.

Die Anpassung an den Kontext zusammengesetzter Komponentensysteme besteht nun darin, dass Prozesse als interagierende Komponenten eines separaten, einstufigen Aggregats *interpretiert* werden.

Die zu synchronisierenden Prozesse bzw. die durch sie modellierten Komponenten werden dann wiederumn in verwendete Teilkomponenten und Toplevelkomponenten, also im- und exportierte Funktionalität, untergliedert.

Die *duale* Interpretation von Prozessen eröffnet die Möglichkeit einer universellen, integrativen Behandlung von Spezifikationen verschiedener Modellaspekte sowie deren gemeinsame Verifikation. In diesem Sinne kann dann der Synchronisator als *lokale* Toplevelkomponente innerhalb eines Komponentenaggregats betrachtet werden, das wiederum Teil einer globalen Struktur sein kann. Der modifizierte Ansatz ist in Abbildung 2.22-B illustriert.

Abbildung 2.22
Modifikation des Spezifikationsmodells



Das so skizzierte Beschreibungsschema erlaubt auch *Teilspezifikationen*. Auf diese Weise kann der vorausgesetzten Annahme begegnet werden, dass konkrete Funktionalitäten von später verwendeten Teilkomponenten zum Entwurfs- bzw. Spezifikationszeitpunkt der Toplevelkomponente noch nicht bekannt sind.

Zum Entwurfszeitpunkt erlaubt der Ansatz eine Spezifikation der Komponentenfunktionalität aus externer Sicht. Dadurch wird in dieser Phase eine Verifikation der Konsistenz in Bezug auf interne Synchronisationsbedingungen sowie Relationen zu externen Funktionalitäten möglich. Soll später eine konkrete Komposition der Toplevelkomponente mit Teilkomponenten erfolgen, erlaubt das Modell den Abgleich derer Spezifikationen zum Kompositionszeitpunkt (vgl. a. Abschnitt 3.10).

Beim Ansatz von Arapis wird die Spezifikation selber in zwei Teile bezüglich Export und Import gegliedert. Diese beiden Teilspezifikationen können als *Schnittstellen* gedeutet werden, und es folgt somit, dass der Ansatz von genau einer Im- und Exportschnittstelle ausgeht. Diese Tatsache führt in zweierlei Hinsicht zu einer unvorteilhaften Spezifikationsstruktur: Zunächst werden Aussagen über verschiedene integrierte Teilkomponenten miteinander vermascht. Des weiteren ist keine eindeutige Unterteilung exportierter Funktionalität in separate Aspekte gegeben, die den multiplen Exportschnittstellen des Komponentenmodells entspricht.

Wie die Ausführungen verdeutlichen, ist der Grundansatz modifizierter Prozeßsynchronisation mit Unterteilung in genau einen Nutzer und n konstituierende Teile auch im Komponentenkontext ein angemessenes Konzept. Potential für Verbesserungen besteht allerdings durch eine Restrukturierung der Spezifikation, um wesentliche Strukturaspekte der Modellebene (s. Abschnitt 2.3.2) adäquater abbilden zu können.

Restrukturierung der Spezifikationsstruktur Die Spezifikation einzelner Verhaltensaspekte geschieht nach obigem Ansatz durch das Aufstellen von PLTL-Formeln. Die verwendeten Propositionen entsprechen dabei den gesendeten bzw. empfangenen Nachrichten bezüglich eindeutig spezifizierter Interaktionspartner. Da solche Formeln durch Konjunktion verknüpft werden können, ist es möglich, sie bezüglich verschiedener Eigenschaften in Teilmengen zu untergliedern. Die Aufgabe besteht nun darin, eine im Komponentenkontext sinnvolle Strukturierung der Einzelaspekte vorzunehmen.

Für die Strukturierung der Einzelaspekte von Komponentenspezifikationen können drei Kriterien angegeben werden:

- **Im- / Export** – Aspekte sollten nach *im-* und *exportierter Funktionalität* getrennt werden. Das bedeutet, Formeln, die entweder nur Nachrichten bezüglich des Nutzers oder der Konstituenten eines Aggregats beinhalten, zu separieren.

- **Schnittstellen** – Der modellhafte *Schnittstellenbegriff* sollte innerhalb der Spezifikation ein Pendant besitzen. Dabei sind zum einen *Importschnittstellen* zu beachten, welche sich in natürlicher Weise durch die verschiedenen Teilkomponenten abspalten, zum anderen sollten aber auch verschiedene *Exportschnittstellen* mit naturgemäß nur organisatorischem Charakter berücksichtigt werden.

- **Interne Synchronisation** – Wechselwirkungen der separierten Teilaspekte sollten durch explizite *Synchronisationsaspekte* zusammengefaßt werden.

Ein Ansatz für die konkrete formale Spezifikation einer Komponente nach obigem Ansatz ist die in Abbildung 2.23 gezeigte Idee einer exportorientierten Strukturierung.

Als Grundprinzip erfolgt hier eine Zerlegung nach Exportschnittstellen. Die Spezifikation der Exportschnittstellen wird dann in die folgenden drei Bereiche unterteilt:

- **Lokal:** Beschreibung der Exportschnittstelle ausschließlich in Bezug auf die eigenen Bestandteile.

- **Export:** Beschreibung der Exportschnittstelle bezüglich ihrer Abhängigkeiten zu den n anderen Exportschnittstellen.

- **Import:** Beschreibung der Exportschnittstelle bezüglich ihrer Verwendung von importierter Funktionalität der m Teilkomponenten.

Ein auffälliges Merkmal dieses Ansatzes ist die Unterordnung von Aspekten genutzter Teilkomponenten bezüglich der Exportaspekte. Dies ist aber im Komponentenkontext plausibel, denn unter methodologischen Gesichtspunkten komponentenbasierter Systeme ergibt sich, wie in Abschnitt 2.3 gesehen, die Ableitung benötigter Teilfunktionalität aus den Vorgaben der angestrebten Exportfunktionalität.

Rolle und Problematik der Propositionen Propositionen bzw. Aussagen sind die grundlegenden Einheiten der propositionalen temporalen Logik. Die angestrebte Logik soll es erlauben, Aussagen über die Anwendungssemantik der funktionalen Einheiten von Komponenten zu machen. Die Aussagen der Logik müssen daher mit eben diesen funktionalen Einheiten in eineindeutiger Relation stehen.

Gesamtspez. $=$	Export 1 $\parallel$	$\wedge \quad \ldots \quad \wedge$	Export n $\parallel$
Lokal	E_1^l	$\wedge \quad \ldots \quad \wedge$	E_n^l
$\wedge$			
Export	$E_1^{e_1}$ $\vdots$ $\wedge$ $\vdots$ $E_1^{e_n}$	$\wedge \quad \ldots \quad \wedge$	$E_n^{e_1}$ $\vdots$ $\wedge$ $\vdots$ $E_n^{e_n}$
$\wedge$			
Import	$E_1^{i_1}$ $\vdots$ $\wedge$ $\vdots$ $E_1^{i_m}$	$\wedge \quad \ldots \quad \wedge$	$E_n^{i_1}$ $\vdots$ $\wedge$ $\vdots$ $E_n^{i_m}$

Abbildung 2.23 *Exportbasierte Spezifikation*

Die wichtigste Anwendung des formalen Apparats besteht in der Verifikation von Verträglichkeit logischer Formeln – bzw. damit der Anwendungssemantik – verschiedener Komponenten. Diese Verifikation setzt eine übereinstimmende Interpretation der verwendeten Propositionen als Modellbestandteile – genauer Nachrichten der Modellebene – voraus. Um letztendlich mit Formeln arbeiten zu können, bedeutet das, dass die Syntax von Propositionen, welche in unterschiedlichen formalen Ausdrücken für die – aus Modellsicht – identischen funktionalen Einheiten stehen, ebenfalls identisch sein muss.

Eine einfache Lösung dieses Problems sind syntaktische Namenskonventionen, wie sie auch bei dem oben angesprcohenen Ansatz von Arapis zum Einsatz kommen. Zusammengesetzte Komponenten beziehen sich auf Teilkomponenten bzw. deren Nachrichten durch Verwendung syntaktischer Bezeichner, die in den jeweiligen Bausteinen direkte Entsprechungen besitzen. Diese Vorgehensweise ist allerdings nur dann sinnvoll, wenn im

Rahmen der Aufbereitung der betroffenen Anwendungsdomäne ein Fachvokabular standardisiert wurde.[58]

Um den Problemen der namensbasierten Vorgehensweise zu begegnen, sollen in den folgenden Abschnitten Ansätze aus den Bereichen flexibler, logikbasierter Strukturbeschreibungen und Typisierung diskutiert werden, die semantisch aussagekräftigere Angaben erlauben könnten.

2.3.6 Flexible Beschreibungsformen mittels Feature-Logik

Ein genereller Nachteil traditioneller Strukturbeschreibungen ist ihre Starrheit. Strukturelle Eigenschaften werden meist fest in den Systemen kodiert. Als flexibler Ansatz für dynamischere und generische Spezifikationen struktureller Aspekte könnte ein auf der *Feature-Logik* basierendes Konzept dienen. Analog zur Betrachtung der temporalen Logik führt auch hier Anhang I die im Folgenden verwandte formale Notation und Terminologie ein[59] zur Diskussion eines auf der Feature-Logik basierenden Spezifikationskonzeptes an.

2.3.6.1 Feature-Logik basierte Formalisierungskonzepte eines Komponentenmodells

Grundsätzlich erlaubt die Feature-Logik die Auswahl von Objekten[60] anhand einer Beschreibung ihrer *Eigenschaften*. Bezüglich des Komponentenmodells kann diese formale Semantik in zwei Bereichen angewandt werden. Zum einen auf die Selektion von *Struktureinheiten* und zum anderen auf die von *Verhaltensaspekten*. Als Features und somit Selektionskriterien müssen in expliziter Weise geeignete Modelleigenschaften bestimmt werden (s.a. Abschnitt 5.3).

Eine Basisspezifikation Komponentenorientierte Strukturspezifikationen mit Feature-Logik könnten auf Basis des folgenden Schemas verwirklicht werden. Der Ansatz beinhaltet eine globale

[58] Eine projektbezogene oder intuitive Namensvergabe, wie sie sich leider häufig in der praktischen Umsetzung der Objektorientierung findet, ist unbedingt zu vermeiden.

[59] insbesondere die der um ein Rollenkonzept erweiterten deskriptiven Form der Feature-Logik

[60] Es ist wiederum nicht der Begriff aus der Objektorientierung gemeint.

Objektmenge von *Struktureinheiten* und drei wesentliche Features:

- **Objekt**: Art der Struktureinheit
- **Informale_Semantik**: Eindeutige, abstrakte Bedeutung (gemäß einer Domänenanalyse)
- **Formale_Semantik**: Verhaltensspezifikation

»Objekt« bezeichnet die Struktureinheiten, wobei in diesem vereinfachten Konzept der Wertebereich als { Komponente, Attribut, Nachricht, etc } definiert ist. Das zweite Feature »Semantik« teilt sich in eine informale Variante, die für eine explizite Typisierung der internen Bedeutung von Strukturelementen steht und eine formale Variante mit der Beschreibung von zeitlich / kausalen Verhaltensaspekten. Der Wertebereich umfasst hier beliebige Bezeichner bzw. PLTL-Formeln.

Die Objekte sind ferner durch eine Reihe von *bidirektionalen Beziehungen* verknüpft, wobei stets Komponenten als zentraler Bezugspunkt involviert sind. Dieser zweiseitige Ansatz erleichtert später den Umgang mit den Spezifikationen, da er die Selektion ausgehend von einer beliebigen Seite (eines beliebigen Interaktionspartners) ermöglicht. Folgende Beziehungen sind bezüglich des zentralen Komponentenkerns enthalten:

- **Export**: Menge exportierter Interaktionseinheiten (=Nachrichten)
- **Zustand**: Attributmenge, die den Komponentenzustand abstrahiert
- **Verhalten**: Menge der Verhaltensaspekte
- **Teil**: Toplevelkomponente bzw. Teilkomponentenmenge

Zur Veranschaulichung zeigt Abbildung 2.24 das Spezifikationsschema am Beispiel der schon früher gezeigten Maschinenkomponente.

Es ist nun möglich, mit Hilfe der Feature-Logik Teile einer Komponente *dynamisch zu selektieren*. Die Nachrichtenmenge einer gegebenen Komponente kann demnach wie folgt umschrieben werden:[61]

$$A = \left[\begin{array}{l} \exists\, Objekt(Nachricht), \\ Export_von : \exists\, Informale_Semantik(Maschine) \end{array} \right]$$

[61] vgl a. den Begriff der Domänenaktivität bzw. des Business Event in Kapitel 4

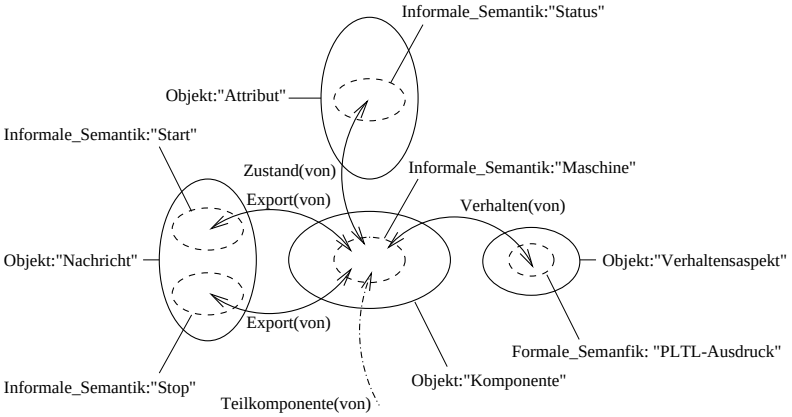


Abbildung 2.24 *Anwendung der Feature Logik zur Komponentenstrukturbeschreibung*

Es ist so ebenfalls möglich, zu einem gewünschten – also gegebenen – Verhalten die Menge aller Komponenten zu finden, welche diesem genügen. Der entspreche Ausdruck lautet dann:

$$A = \left[\begin{array}{l} \exists\, Objekt(Komponente), \\ Verhalten : \exists\, Formale_Semantik(Wunschverhalten) \end{array} \right]$$

Wie diese Beispiele zeigen, ist eine Spezifikation nach dem vorgestellten Konzept durch seine dynamischen Beziehungen weitaus weniger starr als herkömmliche Systeme. Des weiteren beinhaltet der Ansatz eine erste, wenn auch noch separierte Integration von Strukturspezifikation und Verhaltensbeschreibung durch die Kombination der beiden Logiken.

Verfeinerte Beschreibungsansätze Potenzielle Verfeinerungen des oben gezeigten Ansatzes bieten sich vor allem in zwei Bereichen an. Zum einen kann die *interne Struktur* von Komponenten detaillierter modelliert werden, zum anderen sind die *Beziehungen zwischen Komponenten* genauer abbildbar, wobei im Rahmen der vorliegenden Arbeit im wesentlichen letzteres im Mittelpunkt des Interesses steht.

Bezüglich des ersten Aspektes bieten sich sofort *Schnittstellen* als zusätzliche Struktureinheiten an. Des weiteren könnten aber auch beliebige andere Einheiten wie etwa *Kanäle* berücksichtigt werden. Da *Verhaltensaspekte* einer Komponente meist nur teilweise – etwa bei der Verifikation einer Komposition – benötigt werden, scheint es an dieser Stelle sinnvoll zu sein, eine Untergliederung bezüglich der gewählten Strukturebenen vorzunehmen. Ein entsprechendes Spezifikationsschema ist in Abbildung 2.25 gezeigt.

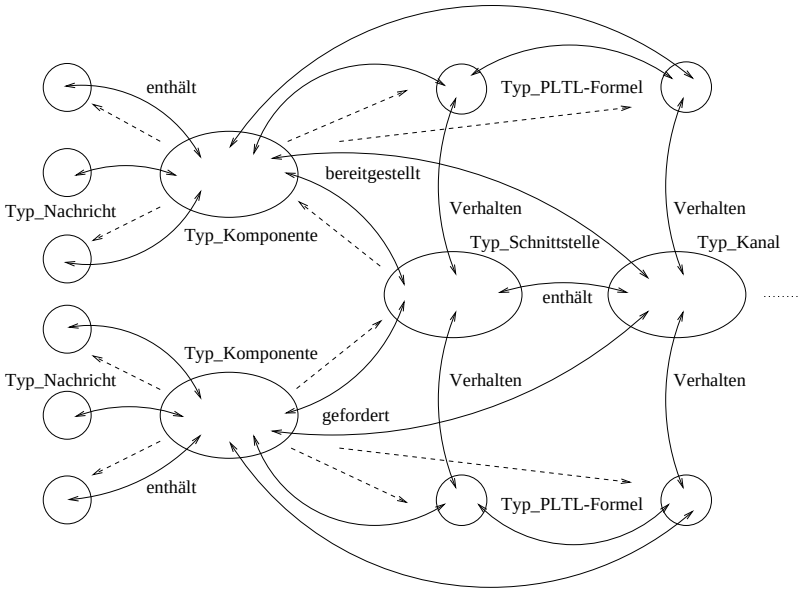


Abbildung 2.25
Multi-Level Modell mit beliebigen Strukturen

Die Strukturierung der Komponente bezieht in diesem Fall zwei weitere Ebenen, nämlich *Schnittstellen* und – exemplarisch für eine beliebige Abstraktion stehend – *Kanäle* mit ein. Letztere Ebenen stellen allerdings formal gesehen keine Erweiterung der Ausdrucksmächtigkeit dar, denn die Verhaltensspezifikation mittels PLTL berücksichtigt weiterhin die atomaren Nachrichten. Aus diesem Grund nehmen Nachrichten in dem Ansatz eine Sonderstellung ein, indem sie in unmittelbarer Beziehung zur Komponenteneinheit stehen. *Kompositionsbeziehungen* zwischen Komponenten werden in diesem Ansatz *indirekt* berücksichtigt. Komponenten referenzieren die Strukturebenen jeweils als geforderte oder bereitgestellte Einheiten und liefern zu diesen eine spezifische Form der Verhaltensbeschreibung. Mittels Feature-Logik kann nun zu einer Komponente nicht nur deren Strukturierung in einer Applikation flexibel selektiert werden, sondern es ist auch ein Retrieval aller potenziellen Gegenstücke möglich. Zu diesen liegen dann auch unmittelbar die relevanten Teile ihrer Verhaltensbeschreibungen vor, welche sofort mit den eigenen Gegenstücken abgeglichen werden können.

Um die gewünschten Informationen durch Feature-Logik zu ermitteln, kann eine sukzessive Vorgehensweise angewandt werden. Für eine Objektmenge mit den Features »Objektart« und »Semantik« sind die geforderten Importschnittstellen einer be-

kannten Komponente wie folgt bestimmbar:

$$I = \exists\, gefordert_von\,(Semantik : \check{}\,Komponentenname\,')$$

Mögliche Gegenstücke werden dann mit dem nachfolgenden Ausdruck gefunden:

$$B = \exists\, bereitgestellt\,(Semantik : \check{}\,Schnittstellenname\,')$$

Sollen nun die zeitlichen Verhaltensaspekte abgeglichen werden, müssen die PLTL-Ausdrücke der Schnittstellenebene sowohl von Bereitsteller- als auch Nutzerseite bestimmt werden. Die entsprechende Formel der Feature-Logik lautet:

$$F = \exists\, Verhalten_von \left(\left[\begin{array}{l} Semantik : \check{}\,Schnittstelle1\,', \\ \left\{ \begin{array}{l} \exists\, gefordert(Semantik : \check{}\,Komponente1\,'), \\ \exists\, bereitgestellt(Semantik : \check{}\,Komponente2\,') \end{array} \right\} \end{array} \right] \right)$$

Bei Interesse an weiteren Strukturstufen kann schließlich mit dem nächsten Ausdruck die Menge von in der aktuellen Stufe befindlichen Untereinheiten selektiert werden:

$$L = \left\{ \begin{array}{l} \forall\, enthalten_in \\ \left(\left[\begin{array}{l} Semantik : \check{}\,Schnittstelle1\,', \\ \exists\, gefordert(Semantik : \check{}\,Komponente1\,') \end{array} \right] \right), \\ \\ \forall\, enthalten_in \\ \left(\left[\begin{array}{l} Semantik : \check{}\,Schnittstelle1\,', \\ \exists\, bereitgestellt(Semantik : \check{}\,Komponente2\,') \end{array} \right] \right) \end{array} \right\}$$

Für weitere Ebenen sowie die jeweiligen spezifischen Verhaltensaspekte gelten die gleichen Formeln wie oben.

Der nächste konsequente Schritt einer weiterführenden Konzeptionierung ist die *Multilevel-Spezifikation*, welche interne Strukturierungsbegriffe von Komponenten gänzlich offen lässt. Die Grundidee ist dabei eine Abstraktion von konkreten Strukturelementen zwischen den Grenzen von Komponente und funktionaler bzw. kommunikativer Basiseinheit (im Allgemeinen Nachrichten). Verschiedene Level wie Kanäle, Prozeduren, Methoden oder Schnittstellen ergeben sich dann aus den konkreten Beziehungen, welche durch Feature Logik formuliert werden. Jeder Level kann dabei wieder seine eigenen semantischen Constraints enthalten, welche zu Vergleichszwecken durch Retrieval-Mechanismen der Feature-Logik konzentriert und vereint werden. Die Constraints in temporaler Logik sind dabei durch ihren Bezug auf eindeutige Basiseinheiten der Kommunikation nicht von diese Ebenen abhängig.

2.3.7 Übergreifende Aspekte formaler Konzepte

Bei den bisherigen Betrachtungen des Themengebietes Spezifikation wurden Ansätze zu verschiedenen Aspekten wie der Beschreibung kausalen / zeitlichen Verhaltens oder der flexiblen Strukturbeschreibung behandelt. Die Ausführungen konzentrierten sich dabei jeweils auf den separierten Aspekt, ohne weiter auf *Wechselwirkungen* einzugehen. Eben diese Wechselwirkungen, seien sie nun positiver oder negativer Natur, sollten aber ebenfalls beachtet werden. Letztere sind insbesondere dann von Interesse, wenn Einzelbetrachtungen Defizite aufweisen, bei denen die Synergie eines integralen Konzeptes auf Lösungsansätze hoffen lässt. Im Folgenden wird zunächst die zum Teil schon angedeutete Kombination der logikbasierten Spezifikationsansätze betrachtet. Als klassische Form übergreifender, integrierender Formalisierung schließen sich Fragen bezüglich möglicher *Typisierungen* an.

2.3.7.1 Kombinationsmöglichkeiten deskriptiver und temporaler Logiken

Wie bereits gesehen, scheint durch temporale Logik alleine keine zufriedenstellende Komponentenbeschreibung möglich zu sein, denn es kann zwar eine aussagekräftige, kausale Verhaltensbeschreibung formuliert werden, das Festmachen derer Gegenstände – also der agierenden Strukturen – ist aber nicht in ausreichend flexibler Weise möglich. Genau an dieser Stelle könnte nun die deskriptive Feature-Logik ihren Beitrag leisten. Abbildung 2.26 veranschaulicht nochmals die Rollen der Logiken in einer Komponentenspezifikation.

Die kausalen Folgen abstrakter Propositionen des Formalismus erhalten im Modell die Bedeutung von *Kommunikationsprotokollen* bezüglich atomarer, externer Interaktionseinheiten (vgl. a. Abschnitt 4.3). Gleichzeitig wird der fehlende Bezug zur strukturellen Semantik deutlich, denn die verschiedenartigen Modelleinheiten werden durch Propositionen abstrahiert, welche ihrer Natur nach gleichartig sind und keine Differenzierung erlauben. Dabei tritt die Situation auf, dass syntaktisch divergente Propositionen Abbildungen identischer Modelleinheiten darstellen. Für ein strukturelles Matching fehlt dann die Informationsgrundlage. Genau in diesem Strukturbezug temporaler Constraints liegt der Berührungspunkt zur deskriptiven Feature-Logik. Eine Kombination der beiden Logiken birgt die Hoffnung verbesserter Bausteinbeschreibungen, denn sie könnte

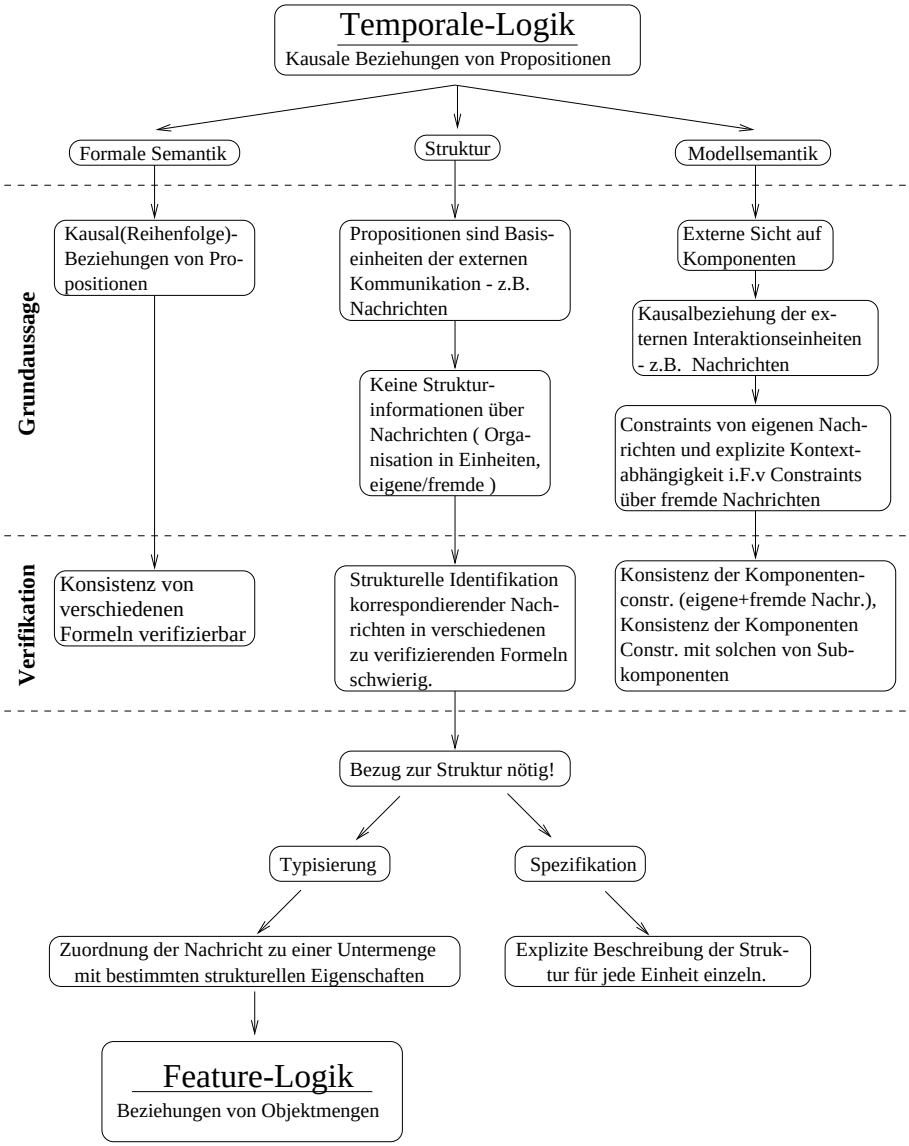


Abbildung 2.26
Rollenverteilung der Logiken

es möglich machen, die Semantik von Verhalten und Struktur zu verbinden.

Es stellt sich dann die Frage nach den *Integrationsmöglichkeiten* der Formalismen. Grundsätzlich gäbe es dabei zwei Varianten zur Kombination von Feature- und temporaler Logik:

- **Beschreibung temporaler Constraints mit Propositionen in Form von deskriptiven Feature-Termen.**

Der erste Ansatz besteht darin, die bisherigen temporalen Ausdrücke um aussagekräftigere Propositionen zu erweitern. Für den Ausdruck

$$\Box\,(Start \Rightarrow \Delta\, Stop),$$

würde durch Ersetzung der Propositionen durch Feature-Terme nach dem Basisschema der obigen Diskussion etwa zu folgendem ersten Ergebnis resultieren:

$$\Box\,([Semantik : \text{»Start«}] \Rightarrow \Delta\, [Semantik : \text{»Stop«}])$$

Die Propositionen haben nun statt der syntaktischen eine formale Identität erhalten. Hierdurch ist jedoch nichts gewonnen, denn die »formale Identität« der Feature-Terme beruht letztlich wieder auf den syntaktischen Bezeichnern, die den Wertebereich der Features ausmachen!

Ein gewisser positiver Effekt lässt sich dennoch erwirken: Propositionen können etwas ausführlicher etwa wie folgt beschrieben werden:

$$\Box \left(\begin{array}{c} \left[\begin{array}{l} Objekt : \text{»Nachricht«}, \\ Semantik : \text{»Start«}, \\ \exists\, Import_von(Semantik : \text{»Maschine«}) \end{array} \right] \\ \Rightarrow\ \Delta \\ \left[\begin{array}{l} Objekt : \text{»Nachricht«}, \\ Semantik : \text{»Stop«}, \\ \exists\, Import_von(Semantik : \text{»Maschine«}) \end{array} \right] \end{array} \right)$$

Durch den Bezug auf eine gegebene Komponente schränkt sich der *Namensraum* für Nachrichtenbezeichner auf diese ein (vgl. a. Abschnitt 5.3). Dadurch wird ein anschließendes Matching zumindest erleichtert.

- **Beschreibung von Mengen struktureller Modellbestandteile durch Feature-Logik anhand temporaler Constraints als Features.**

  Der zweite Integrationsansatz sieht vor, Features zu definieren, deren Wertebereich Ausdrücke temporaler Logik sind. Das Ergebnis ist aber noch unbefriedigender als im vorherigen Fall, denn ein zur Selektion nötiger Vergleich von Features entspricht genau dem gleichen Vorgang wie bei der herkömmlichen Spezifikationsform. Die Problematik anonymer Propositionen bleibt gänzlich unberührt und wird nur in eine andere Ebene verlagert.

Nach längerer intensiver Untersuchung dieser »Matching«-Problematik äquivalenter funktionaler Einheiten verschiedener Komponenten im Rahmen dieser Arbeit bleibt als Ergebnis, dass es zu einer expliziten und einheitlichen *Benennung* keine Alternative gibt. Durch Versuche, funktionale Einheiten mit Hilfe von Feature-Logik abstrakt zu umschreiben, können die erhofften Resultate wie gesehen nicht in vollständig befriedigender Form erreicht werden. Da durch den Einsatz der Logik dennoch eine signifikante semantische »Anreicherung« einer Komponentenspezifikation erreichbar ist, sollen beide Aspekte beim Entwurf des Klassifikationskonzeptes in Kapitel 4 berücksichtigt werden.

Die Problematik liegt darin, dass das Wesen des Unterschieds verschiedener funktionaler Einheiten aus einer Außensicht nicht formal erfasst werden kann. Dieses Wesen liegt in der den Einheiten *innewohnenden* Semantik. Letztere wird in vollständiger Weise letztendlich nur durch deren Implementation beschrieben, welche aber durch eine Außensicht gekapselt werden soll. Kapselung impliziert stets eine, wenn auch frei wählbare Grenze externer Einblicke. Ab dieser Grenze – der *Abstraktionsbarriere* – lässt sich die Semantik nur durch explizite Benennung unterscheiden.

Wenn berechtigterweise globale *Namenskonventionen* aus pragmatischen Gründen ausgeschlossen werden, so muss zumindest auf *semiautomatische* oder gar komplett *benutzergetragene* Matching-Mechanismen zurückgegeriffen werden. Es kommen dann zum Beispiel explizit vom Benutzer definierte Beziehungen in Frage, welche in Repositories gehalten und bei Bedarf abgefragt werden können. Dies hat den Autor bewogen ein *explizites Typmanagement* als zentrales Konzept im Rahmen einer komponentenbasierten Sichtweise zu fordern. Ein entspechendes Klassifikationkonzept wird in Kapitel 4 vorgestellt. Im Folgenden soll aber bereits vorab kurz auf entsprechende Ansätze eingegangen werden, da ihr Verhältnis zu Spezifikationskonzepten zu klären ist.

2.3.7.2 Typisierungsansätze

Um sich den Problembereichen der bisher behandelten Formalisierungsansätze anzunähern, bietet sich als denkbarer Ansatzpunkt traditionell die ergänzende Verwendung von Typisierung an (siehe zum Beispiel [Nierstrasz, 1995a]). Nicht zuletzt deuteten auch die Ergebnisse des letzten Abschnitts auf (explizite) Typisierung als ein potenziell weiterführendes bzw. unumgänglich notwendiges Konzept hin. Angewendet auf die Problematik strukturellen Informationsmangels temporallogischer Aus-

drücke, bietet sich zunächst eine Anwendung des Prinzips auf Propositionen an.

Propositionen könnten demnach neben den komponenteninternen Bezeichnern noch deren *Typinformationen* beinhalten. Das hieße, in expliziter Weise global eindeutige Klassen mit semantisch als gleich erachteten Propositionen bzw. Nachrichten einzuführen. Der Semantikbegriff bezöge sich dabei auf die interne Bedeutung von Einheiten unterhalb der Abstraktionsbarriere, nicht jedoch auf die Verhaltenssemantik. Der Nutzen von Typisierung ergibt sich durch Einführung von *Typbeziehungen*, welche den Rückschluß auf Eigenschaften der entsprechenden Instanzen erlauben, wobei die Grundlage der Beziehungen unerheblich ist. Die wesentlichen Beziehungsformen sind dabei *Konformität* und *Kompatibilität*.[62]

Durch die Verwendung solcher Typen funktionaler Einheiten alleine wäre aber nach wie vor keine eineindeutige Zuordnung verschiedener Bezeichner-Typ-Paare möglich, da Typen immer für eine ganze Menge von Instanzen stehen. Käme es etwa zu der Situation, dass kompatible Typen in einer Formel mehrmals, in einer anderen hingegen nur einmal aufträten, würde für das Matching der Propositionen keine weitere Informationsgrundlage existieren. Diese Tatsache gilt dabei nicht nur für das mehrfache Auftreten eines Typs, sondern ergibt sich schon für konforme Arten.

Es stellt sich in diesem Zusammenhang die Frage, ob, wie hier geschehen, die Logik typisiert werden soll, oder ob nicht umgekehrt Typrepräsentationen logische Aussagen *enthalten* könnten. In diesem Fall würde die Typisierung auf Semantik der Komponentenebene beruhen. Es muss also geklärt werden, wie die beiden Formalismen *(statische) Typisierung* und *(semantische) Spezifikation* in Zusammenhang gebracht werden sollten.

Als zweite Alternative für Typisierungsansätze sind in diesem Sinne logikbasierte Typen zu betrachten. Zunächst scheint sich dabei die Integration temporaler Constraints in ein Typsystem als natürliche Lösung anzubieten. Die Idee dabei ist eine Typisierung auf Grundlage von PLTL-Spezifikationen, wie sie auch in [Arapis, 1995] und [Nierstrasz, 1995a] angedacht wurde. Komponenten (-Schnittstellen) werden dabei auf Grund der kausalen Beziehungen ihrer ein- und ausgehenden Nachrichten typisiert. Die Spezifikation dieser Verhaltensaspekte erfolgt mit Hilfe von PLTL-Formeln. Der strukturellen Klassifizierung von Typen

[62] Die exakten Definitionen dieser Begriffe sind Gegenstand der folgenden Kapitel.

werden in diesem Sinne einfach temporale Constraints eben dieser Strukturtypen hinzugegeben, und jede Instanz ist damit verpflichtet, ihnen zu genügen. Ein Typmanagement kann dann automatisch eine Typkonformität überprüfen. Matchingprobleme treten nicht auf, da alle Propositionen eindeutige Typbezeichner einer globalen Typmenge sind. So verwendet ein Komponententyp zum Beispiel die Funktionalität einer »Subkomponente«, indem er deren global eindeutige Typinformation referenziert und der Nomenklatur seiner Spezifikation zugrunde legt. Leider gilt auch für diesen Ansatz die Tatsache, dass er zwar für Instanzen des gleichen Typs ein kompatibles Verhalten im Rahmen der Spezifikation garantiert, jedoch keine eineindeutige Identität sicherstellt, da Typen eben immer für ganze Klassen von Instanzen stehen, die sich auf detaillierterer Ebene durchaus unterscheiden können.[63]

Am Ende dieses Abschnittes soll noch kurz die Thematik von Typbeziehungen im speziellen Fall PLTL-basierter Typen angesprochen werden. Bezüglich sinnvoller Typbeziehungen können die Ergebnisse von [Nierstrasz, 1995a] als Ansatzpunkt genommen werden. Nierstrasz unterscheidet folgende Relationen:

- Typkonformität oder -substituierbarkeit
- Typbefriedigung oder -kompatibilität

Es stellt sich dann aber die Frage, wie die Verifikation von PLTL-Spezifikationen auf Konformität (siehe oben oder [Arapis, 1995]) in Beziehung zur Verifikation von Typbeziehungen steht. So ist die Existenz mindestens eines gemeinsamen mathematisch logischen Modells (einer Ausführungsfolge, die den temporalen Constraints beider Vergleichskandidaten genügt) sicher kein geeignetes Kriterium für eine Typkonformität. Vielmehr müsste hier das Kriterium von [Nierstrasz, 1995a] angewandt werden, das besagt, dass ein Typ A durch einen zweiten Typ B austauschbar ist, wenn folgende Bedingungen erfüllt sind:

[63] Das in Kapitel 4 vorgestellte Typmanagement erhält daher entsprechende Identitäten aufrecht, indem es sie neben den zugehörigen Typinformationen gesondert verwaltet. Für den hier verfolgten generativen Ansatz ist die Identitätsfrage allerdings in dem Sinne von untergeordneter Bedeutung, als dass eine Trading-gemäße [Müller et al., 1994] Anfragebearbeitung mit entsprechenden, attributierten Rückgabelisten möglich ist - vgl. a. Kapitel 5.

- Alle endlichen, gültigen Ausführungsfolgen (Modelle[64]) des Typs A sind auch Modelle des Typs B.

- Alle Propositionen, welche im Anschluß an ein beliebiges Modell des Typs A für diesen gültig sind, sind auch für den Typ B im Anschluß an dasselbe Modell gültig.

2.3.7.3 Überblick und Bewertung der Ergebnisse

Nachdem nun einige formale Konzepte zur Spezifikation einer Komponente bzw. aus solchen aufgebauten Strukturen diskutiert wurden, soll im Folgenden eine kurze Bewertung der Ergebnisse vorgenommen werden.

Die behandelten Konzepte umfassten Formalismen zur semantisch reichen Beschreibung von Komponenten. Zunächst wurde ein auf temporaler Logik basierender Ansatz zur Spezifikation von Kausalbeziehungen betrachtet. Dieser erlaubt eine natürliche Formulierung bereitgestellter und erwünschter Eigenschaften von Bausteinen. Auf diese Weise dargestellte Verhaltensmuster konnten auf gegenseitige Verträglichkeit bzw. Konformität geprüft werden, wobei zu diesem Zweck effektive Algorithmen bestehen. Der zweite sich auf der Feature-Logik gründende Ansatz, hat dynamische Spezifikationsmöglichkeiten von internen und externen (d.h. innerhalb der Gesamtstruktur einer Applikation bzw. Top-Level-Komponente) Komponentenstrukturen ermöglicht. Die typisierte Feature-Logik erlaubt die Auswahl beliebiger Struktureinheiten anhand derer Eigenschaften. Beschreibungen von Bausteinen sind auf diese Weise relativ zu deren in- und externen Merkmalen möglich und dadurch dynamisch und flexibel anstatt wie bei herkömmlichen Beschreibungen – etwa in IDL – fest kodiert und entsprechend starr.

Neben dem attraktiven Potential der Ansätze zeigten sich leider auch explizite Grenzen der Anwendbarkeit. Die Problematik ließ sich dabei an einem Zielkonflikt zwischen Abstraktion und Spezifikation festmachen. Während Komponenten einerseits durch Spezifikation in generischer Weise eindeutig in Struktur und funktionalem Verhalten beschrieben werden sollen, werden andererseits durch Abstraktionsbarrieren Grenzen dieser Beschreibung vorgegeben, welche dann ein – bezüglich des Komponentensystems – global einheitliches Verständnis der Abstraktionen voraussetzen. Eben dieses global einheitliche Verständnis fordert die verstärkte Berücksichtigung der Methodik

[64] Der Begriff wird an dieser Stelle im mathematischen Sinne verwendet.

des Domain Engineering (s.a. Abschnitt 5.3), das daher für den konstruktiven Ansatz dieser Arbeit zu Grunde gelegt wird.

Während sich die oben diskutierten Formalismen zwar zur Spezifikation einer Komponentencharakteristik eignen, ist es schwierig auf ihrer Basis angleichende, kompensierende oder sonstige *adaptierende* Mechanismen für ein Komponenten*verhalten* abzuleiten, da beispielsweise durch eine Unifikation logischer Ausdrücke zwar ein *gemeinsames* Modell hergeleitet werden kann, dieses jedoch nicht mehr direkt erkennbar werden lässt, welche ursprünglichen Konzepte eine (gegenseitige) Abbildung erfahren haben. Daher soll für die speziellen Belange einer Adaption im folgenden Kapitel nun zunächst ein *operationales* Adaptionskonzept für das gezeigte grundlegende Nachrichtenbasierte Komponentenmodell entworfen werden.

3 Adaption

Anpassungsfähigkeit bzw. Flexibilität von Komponenten ist eine wesentliche Voraussetzung für deren Wiederverwendbarkeit, da speziell im Fall komplexer Komponenten die detaillierten Umstände einer erneuten Verwendung in fremden Systemen nicht komplett übersehen werden können oder so mannigfaltig sind, dass eine Berücksichtigung aller Möglichkeiten zu einer übergroßen Menge sehr ähnlicher Komponenten führen würde. Insbesondere im Falle einer automatisierten Komposition können entsprechende Adaptionsmechanismen eine hilfreiche Abstraktion der einzelnen Komponentenverbindungen sein. Ein solcher *Adapter* kann dann als aktive Komponente im Gegensatz zu dem im vorangegangenen Kapitel diskutierten formalen Auswertungen *operational* arbeiten. Damit wird eine komponentenbasierte Struktur, wie sie auch Gegenstand der vorangegangenen Betrachtungen war, um ein zusätzliches Konzept angereichert, indem statt einer direkten Kopplung explizite »Verbindungsstücke« zwischen Komponenten in Erscheinung treten. Um ihre Sonderrolle zu kennzeichnen sollen diese Komponenten im weiteren als *Konnektoren* bezeichnet werden.

3.1 Konzepte statischer und dynamischer Komponentenadaption

Die Komponentenkopplung stellt den grundlegenden Mechanismus dar, mit dem Komponenten sowohl in der Kompositionsphase als auch in der Laufphase (statisch bzw. dynamisch) Verbindungen miteinander zum Zwecke der Interaktion eingehen können. Reibungslose und fehlerfreie Komponentenkopplung ist somit der ausdrückliche Wunsch von Anwendungsentwicklern, die ein System mittels komponentenbasierter Softwareentwicklung implementieren möchten.

Aufgrund potenziell unterschiedlicher Herkunft (mit unterschiedlichen Implementierungsprämissen) der dem Entwickler zur Verfügung stehenden Komponenten, ist eine reibungslose

Komponentenkopplung aber unter Umständen nicht trivial. Im Folgenden sollen daher zunächst Bedingungen für die Eigenschaft der »Kompatibilität« zweier Komponenten im Sinne ihrer konkreten Kopplungsfähigkeit erörtert werden. Im Folgenden wird das Augenmerk noch einmal näher auf die Konzepte erweiterter Schnittstellenspezifikationen gerichtet, die eine zentrale Bedeutung für alle in dieser Arbeit diskutierten adaptions- und kompatibilitätsprüfenden Konzepte besitzen. Für die Überwindung existierender Inkompatibilitäten zweier Komponenten werden dann mögliche Konzepte der Komponentenadaption vorgestellt.

3.1.1 Komponentenkompatibilität

Komponentenorientierte Anwendungsentwicklung basiert auf der Komposition abgeschlossen implementierter Komponenten. Wenn eine Kopplung von Komponenten in einem spezifischen Kontext möglich ist, werden die Komponenten als »kompatibel« zueinander bezeichnet. Der Begriff Kompatibilität umfasst dabei sowohl den semantischen als auch den syntaktischen Aspekt der Komponentenkopplung.

Aufgrund des unterschiedlichen Charakters beider Aspekte ist eine abstrahierende Aufteilung in den Begriff der *semantischen* und den Begriff der *syntaktischen* Kompatibilität sinnvoll. Folgende Charakterisierungen bieten sich zunächst an:

Definition 3.1 (semantische Kompatibilität)

Komponenten werden als zueinander semantisch kompatibel bezeichnet, wenn sie im Verlaufe der Interaktion miteinander ein gegenseitiges gemeinsames Verständnis des Konzeptes und der Bedeutung der kommunizierten Informationen besitzen. Die dadurch erreichte semantische Kompatibilität bezieht sich auf den Bereich einer konkreten fachspezifischen Anwendungsdomäne.

Semantische Kompatibilität ist somit eine sich lediglich auf konkrete (eben anwendungsspezifische) Voraussetzungen beziehende Eigenschaft, die keine objektive Allgemeingültigkeit unter verschiedenen Bedingungen[1] besitzt. Die Gültigkeit dieser Aussage lässt sich exemplarisch an Komponenten aufzeigen, die in der Interaktion mit anderen Komponenten verschiedene Rollen anzunehmen in der Lage sind. Eine solche Komponente kann ihren Methoden und Parametern je nach der Rolle, die sie im jeweiligen Moment im anwendungsspezifischem Kontext spielt, unter-

[1] etwa in verschiedenen Anwendungsdomänen

schiedliche semantische Bedeutungen verleihen, was ein Auftreten semantischer Inkompatibilität auch zu späteren Zeitpunkten innerhalb der Laufphase der Anwendung ermöglicht.[2]

Definition 3.2 (syntaktische Kompatibilität) *Komponenten werden als zueinander syntaktisch kompatibel bezeichnet, wenn die während der Interaktion der Komponenten ausgetauschten Nachrichten zueinander syntaktisch kompatibel sind. Ein Nachrichtenaustausch zwischen Sender und Empfänger einer Nachricht ist syntaktisch kompatibel, wenn sowohl der Methodenname als auch die Anzahl, die Reihenfolge und die Datentypen gegebenenfalls zugehöriger Paramter beim Sender und beim Empfänger dieser Nachricht exakt übereinstimmen (also die Gestaltung der Schnittstellen den Nachrichtenaustausch zur Interaktion erlaubt).*[3] *Darüber hinaus dürfen im Kommunikationsablauf keine Störungen in der Art unspezifiziertem Empfangs oder Verklemmungen (engl. deadlocks) auftreten.*[4]

Syntaktische Inkompatibilität lässt keine fehlerfreie Komponentenkopplung zu und sollte von der Systemumgebung bereits während des Kopplungsvorganges erkannt werden und diesen entweder verhindern oder mittels geeigneter Anpassungskonzepte ermöglichen.

Hervorzuheben ist ferner, dass eine semantische Kompatibilität unabhängig von einem syntaktisch inkompatiblen Nachrichtenaustausch vorliegen kann. So ist es beispielsweise für die semantische Bedeutung eines Parameters, der das Lebensalter einer Person als Information enthält, unerheblich, ob dieser Wert nun vom Typ *ganze Zahl* oder vom Typ *positiv ganze Zahl* ist. Syntaktisch hingegen ist es von Bedeutung und es führt zur Inkompatibilität, falls interagierende Komponenten unterschiedliche Datentypen für einen solchen Parameter voraussetzen.

Die Überwindung syntaktischer Inkompatibilität kann somit semantisch kompatiblen Komponenten die Kopplung ermögli-

[2] Aus diesem Grund postuliert der Ansatz dieser Arbeit aktive, auch in der Laufzeitstruktur vorhandene Konnektoren (vgl. auch [Griffel, 1996]). Ferner sollte bereits zum Kompositionszeitpunkt eine Prüfung aller potenziellen Rollen berücksichtigt werden (vgl. a. Abschnitt 4.8).

[3] Hiermit ist der Begriff der Nachricht zunächst konkret eingeschränkt, um einen Bezug zur Implementationspraxis herstellen zu können. Diese Sicht wird im Rahmen des Typmodells in Kapitel 4 wieder erweitert, um den gewünschten Abstraktionsgrad zu erhalten.

[4] Diese Probleme werden im Verlauf des Kapitel noch detailliert behandelt.

chen, die ansonsten nicht fehlerfrei möglich wäre. Eine derartige Überwindung syntaktischer Inkompatibilität ist somit insbesondere Gegenstand der vorliegenden Arbeit.

3.1.2 Komponentenanpassung

Die Anpassungsfähigkeit einer Komponente ist einer der Schlüssel zu ihrer Wiederverwendbarkeit. Konkretes Ziel der Anpassung einer Komponente ist das Erreichen semantischer und syntaktischer Kompatibilität zu den mit ihr interagierenden Komponenten. Der Vorgang des Veränderns einer Komponente zum Zwecke aktueller spezifischer (Wieder-) Verwendung (also Kopplung) mit anderen Komponenten wird als *Anpassung* bezeichnet. Unterschieden wird hierbei nach dem Zeitpunkt der Anpassung zwischen *statischer* und *dynamischer* Anpassung.

3.1.2.1 Statische Anpassung

Statische Anpassung bezeichnet den Anpassungsprozess während der Entwicklungsphase einzelner Komponenten oder der Kompositionsphase der Gesamtanwendung. Methoden hierzu werden in aktuellen Komponentensystemen bereitgestellt. Die variablen Merkmale einer Komponente werden dabei durch Properties (vgl. a. Abschnitt 3.10) repräsentiert, zu deren Modifikation Property-Editoren oder ähnliche Werkzeuge dienen.

3.1.2.2 Dynamische Anpassung

Dynamische Anpassung bezeichnet den Anpassungsprozess während der Laufphase der Gesamtanwendung. Eine solche Laufzeitänderung einer Komponente erscheint besonders attraktiv, da mit dem Begriff der dynamischen Anpassung der Begriff der automatisierten oder *Selbstanpassung* eng verbunden ist.

Optimalerweise würden somit alle benötigten Anpassungsprozesse, deren Notwendigkeit zur Laufzeit auftritt, ohne Eingreifen eines Entwicklers automatisch und transparent ablaufen. Entsprechende Mechanismen der Selbstadaption sind Gegenstand laufender Forschung [Batory, 1997] und berühren auch Fragestellungen dieses Kapitels, werden in dieser Arbeit jedoch nicht schwerpunktmäßig betrachtet (vgl. aber a. Abschnitt 3.10). Auch technisch stellt eine dynamische und automatisierte Komponentenanpassung eine große Herausforderung dar. Grundbedingung ist dabei die Möglichkeit des Austauschens möglichst umfangreicher Metainformationen der Komponenten untereinander.

3.1.2.3 Anpassungsmethoden

Komponentenanpassung kann nicht nur zu verschiedenen Zeitpunkten (s.o.), sondern auch auf verschiedene Art erreicht werden. Entscheidend für das jeweils optimale Mittel der Komponentenanpassung ist dabei vor allem die Art der aufgetretenen Inkompatibilität – abstrahierend unterteilt in semantische und/oder syntaktische Inkompatibilität.[5].

Syntaktische Inkompatibilität, die sich ja lediglich auf die Schnittstellen der beteiligten Komponenten bezieht, kann durch die in Abschnitt 2.1.2.2 bereits angeführten Anpassungsmechanismen ausgeglichen werden:

- Schnittstellenstandardisierung: Dieses Konzept verlangt die Anpassung bereits während der Entwicklungsphase einer Komponente. Statische Schnittstellenbeschreibungen in speziellen *Schnittstellenbeschreibungssprachen* (IDL), nach denen der Komponentenentwickler seine Implementation ausrichten muss, sorgen für eine syntaktische Kompatibilität.

- Schnittstellenüberbrückung: Eine spezielle *Schnittstellentransformationssprache* (ITL) sorgt während der Laufphase einer Komponentenkollaboration für einen Ausgleich zwischen *angebotener* Schnittstelle auf der einen und *vorausgesetzter* Schnittstelle auf der anderen Seite. Die ITL muss dabei detailierte Anweisungen für jede notwendige Transformation möglicher Methodenaufrufe enthalten.

Hinzu tritt bei Einführung aktiver Verbindungskomponenten die

- Interzeption mittels Adapterkomponente: Tritt eine syntaktische Inkompatibilität aufgrund von Inkompatibilität im Ablaufprozess erwarteter und gesendeter Nachrichten auf, so wird eine komplexere Schnittstellenüberbrückung benötigt. Die Transformationsfunktion wird hierbei durch einen Adapter mit eigener »Intelligenz« wahrgenommen, der in der Lage ist, dynamischere Transformationen als die mittels ITL ermöglichten durchzuführen. Eine solche Adapterkomponente bietet zum Beispiel die Möglichkeit, intern Methodenaufrufe und zugehörige Parameter zu speichern und

[5] Der Anpassungsbegriff wird hier also insbesondere nicht für das Setzen bestimmter Komponenteneigenschaften verwendet. Dies ist Gegenstand der Merkmalsauswahl (vgl. Kapitel 5)

den Zeitpunkt eigener Methodenaufrufe an die beteiligten Komponenten – abhängig von internen Zuständen – exakt festzulegen. Dieses Konzept ist eine Erweiterung des oben beschriebenen Konzeptes der Schnittstellenüberbrückung und soll zu expliziten Abgrenzung als Interzeption bezeichnet werden (vgl. a. [Griffel, 1996, Griffel et al., 1996]).

3.1.2.4 Bewertung hinsichtlich des Einsatzzeitpunkts

Im Hinblick auf die Anwendbarkeit der oben aufgeführten Adaptionskonzepte in den verschiedenen Phasen einer Anwendungsentwicklung lässt sich festhalten, dass die Schnittstellenüberbrückung mittels ITL oder Interzeption mittels Adapter die Möglichkeit einer dynamischen Veränderbarkeit (der Adapterkomponente bzw. der ITL-Spezifikation) während der Kompositions- und Laufphase bietet. Im Gegensatz dazu kommt eine Schnittstellenstandardisierung nur in – eigentlich *vor* – der Entwicklungsphase einer Komponente zum Tragen. Insbesondere im Rahmen dieser Arbeit von Bedeutung ist die daraus folgende potenzielle Verwendung dieser Konzepte während der Kopplung der Komponenten in der Kompositions- oder Laufphase einer Anwendung.

3.1.2.5 Bewertung hinsichtlich Einsatzgebiet

Wie gut eignen sich nun die verschiedenen Konzepte zur Überwindung syntaktischer bzw. semantischer Inkompatibilität?

Schnittstellenstandardisierung und Schnittstellenüberbrückung mittels IDL und ITL stellen jeweils einen rein syntaktischen Anpassungsmechanismus dar.[6] Der potenziellen Anpassungsfähigkeit sind bei diesen Konzepten jedoch enge Grenzen gesetzt. Syntaktische Inkompatibilität aufgrund differierender Methodensignaturen lassen sich mittels IDL [Christiansen & Münke, 1995] oder mittels ITL [Czarnecki & Eisenecker, 2000] überwinden. Nicht kompatible Kommunikations*abläufe* – also das eigentliche *Verhalten* von Komponenten – hingegen können mittels dieser Techniken nicht überwunden werden.

[6] Entsprechende Versuche einer semantischen Anreicherung dieser Konzepte, wie etwa mit der »Component Description Language« der OMG haben sich bislang nicht in der Entwicklungspraxis durchsetzen können, da sie eine erhöhte Komplexität und ein formaleres Denken der Entwickler erfordern. Daher geht der Ansatz dieser Arbeit zwar vom Vorliegen umfangreicher Spezifikationen, aber gleichzeitig deren Verbergen vor dem Anwendungsentwickler aus.

Zur Überwindung derartiger Differenzen wird eine Adapterkomponente benötigt, die in der Lage ist

- Methodenaufrufe gegebenenfalls nicht weiterzuleiten
- Methodenaufrufe an eine beteiligte Komponente zu senden, ohne, dass die andere Komponente zuvor etwas gesendet hat
- Parameterwerte zwischenzuspeichern und auszulesen, um derartige kommunikationssteuernde Mechanismen zu ermöglichen.

Diese Möglichkeiten bieten sich einer Adapterkomponente, die einen internen veränderbaren Zustand besitzt, aufgrund dessen sie Aktionen (beispielsweise das Senden einer Nachricht) startet oder Kommunikationsaktionen der beteiligten Komponenten erwartet.[7] Die Zustandsmenge und die Regeln für die Zustandsveränderungen – aufgrund eingegangener Nachrichten, interner Vorgänge oder gesendeter Nachrichten – und daraus gegebenenfalls folgende Aktionen sind innerhalb der Adapterkomponente festgelegt.

Zur Überwindung semantischer Inkompatibilität dienen die folgenden Anpassungsmechanismen:

- Anpassung mittels Properties: Bei vom Komponentenentwickler explizit vorgesehenen Properties einer Komponente können in der Kompositionsphase Werte verändert werden. Dieser Anpassungsmechanismus nimmt Einfluss auf die interne Funktionalität der Komponente und kann diese weitreichend verändern.[8] Damit kann sich auch die semantische Bedeutung auszutauschender Parameter aus Sicht einer Komponente verändern lassen. Welche internen Werte einer Komponente als Properties von aussen veränderbar sind, kann nur in der Entwicklungsphase festgelegt werden. Werteänderungen an diesen Properties finden gewöhnlich in der Kompositionsphase statt; sie sind aber auch in der Laufphase einer Anwendung sinnvoll denkbar.
- Interzeption mittels Adapterkomponente: Dieses bereits vorgestellte Konzept kann neben der Überwindung syntak-

[7] Anmerkung: Sie dient damit der Adaption des von aussen beobachtbaren Verhaltens, nicht etwa der Veränderung der Funktionalität einer Komponente.

[8] Daher sollten solche Veränderungen nur als *angeleitete Konfiguration* zulässig sein (vgl. Kapitel 5).

tischer Inkompatibilität auch der Erzeugung einer funktionalen Zusammenarbeit dienen. Ermöglicht wird dies durch die weitreichenden Kommunikationssteuerungsmöglichkeiten eines aktiven Adapters. Der Adapter sollte somit beliebig komplexe Aktionen speichern und umsetzen können. Die Adapterkomponente kann sowohl statisch als auch dynamisch erzeugt oder verändert werden.[9]

Der Überwindung einer semantischen Inkompatibilität sind natürliche Grenzen gesetzt. Zwei Komponenten, deren angebotene Dienste einander in keinerlei Weise entsprechen, lassen sich nicht sinnvoll semantisch anpassen. Somit wäre auch eine syntaktische Anpassung in solch einem Fall nutzlos.[10]

3.1.2.6 Abschließende Bewertung

Der Mechanismus der Interzeption mittels Adapterkomponente bietet als einziger der hier vorgestellten Konzepte die Möglichkeit zur Überwindung aller Arten syntaktischer Inkompatibilität und in Grenzen auch zur Überwindung semantischer Inkompatibilität. Daher wird im weiteren auf diese Adaptionsmethode das Hauptaugenmerk gelenkt.

Die konkrete Verhaltenspezifikation einer Adapterkomponente hängt eng mit der Schnittstellenspezifikation der zu adaptierenden Komponenten zusammen. Ausgehend von dieser Erkenntnis lässt sich gegebenenfalls das interne Verhalten der Adapterkomponente so spezifizieren, dass sie selbst zu den beteiligten Komponenten kompatibel ist und für eine Kompatibilität des Gesamtaggregats der Komponenten sorgt. Um diese Ableitung eines Adapters so weit wie möglich automatisieren zu können, soll im Anschluss zunächst ein Modell einer operationalen Adaption erarbeitet werden.

---

[9] Die in Kapitel 5 entworfene generative Konstruktion arbeitet dabei sowohl unter Einsatz bestehender Adapterkomponenten als auch der bedarfsweisen Erstellung neuer Adapter während des Konstruktionsprozesses.

[10] Daher erscheint die Einführung eine Betrachtungsebene, auf der Anpassungen bewusst ausgeschlossen werden sinnvoll. Der Entwurf des Klassifikationskonzept im nachfolgenden Kapitel berücksichtigt dies explizit.

3.2 Ein Modell dynamischer Komponentenadaption

Das im Folgenden erörterte Modell entwickelt und untersucht Konzepte protokollbasierter Komponenteninteraktion, dynamischer Kompatibilitätsprüfung und automatisierter Adaption. Die Konzepte dieses Modells erlauben auf der Basis einer erweiterten, operationalen Schnittstellenspezifikation in Anlehnung an [Yellin & Strom, 1995] eine formale Untersuchung potenziell interagierender Komponenten *vor* einem möglichen Kopplungsvorgang im Hinblick auf die Frage, ob eine syntaktisch kompatible Kopplung möglich ist. Zudem wird die semiautomatische Erstellung von Adapterkomponenten ermöglicht, die die Überbrückung etwaiger Inkompatibilitäten der Schnittstellen potenzieller Kopplungspartner mittels Interzeption erlauben. Desweiteren kann die Kompatibilität einer formalen Adapterkomponentenbeschreibung zu den formalen Beschreibungen der mit ihr verbundenen Komponenten eindeutig geprüft werden.

Zentraler Bestandteil dieses Modells ist die formale Angabe exakter Kommunikationsabläufe in einer Schnittstellenbeschreibungssprache. Im Gegensatz zu einfachen Schnittstellenbeschreibungskonzepten wie beispielsweise einer IDL, legt der Komponentenentwickler hier exakt die sinnvoll möglichen Kommunikations*abläufe* einer Komponente fest. Ferner werden alle angebotenen sowie potenziell zu sendenden Nachrichten explizit angegeben.

Der entsprechende Bezug zur Implementationspraxis wird hergestellt, indem die Nachrichten als Methoden(signaturen) konkretisiert werden. In der Weiterführung dieses Konzepts wird dann auch eine Parametrisierung mit einbezogen, d.h. gegebenenfalls auch entsprechende Parameterlisten adaptiert. Dies ist realitätsnah und erhöht die Zuverlässigkeit der Kompatibilitätsprüfung und der möglicherweise folgenden Adaption der interagierenden Komponenten.

3.3 Begriffseinführung und Voraussetzungen

Betrachtet werden gemäß des in Kapitel 2 diskutierten Basismodells Komponenten, die miteinander über Schnittstellen im Sinne eines Nachrichtenaustauschs kommunizieren. Der Austausch von Nachrichten zum Erbringen einer Gesamtfunktiona-

lität wird als Kollaboration bezeichnet. Die Regeln für mögliche Reihenfolgen auszutauschender Nachrichten innerhalb der Kollaboration werden in einem deklarativen Teil jeder Komponente vom Komponentenentwickler festgelegt – der sogenannten *Kollaborationsspezifikation*.

3.3.1 Kollaborationsspezifikation

Eine Kollaborationsspezifikation, die einer Komponente beispielsweise als Metainformation beigefügt werden kann (vgl. Kapitel 6), besteht aus zwei Teilen:

- Strukturelle Schnittstellenspezifikation
- Protokoll

Die *Schnittstellenspezifikation* listet alle Empfangs- und Sendenachrichten mit den zugehörigen Parametern auf, die diese Komponente verarbeiten bzw. emittieren kann.

Das Protokoll wird mit einer Liste bestehend aus einem Anfangszustand und einem oder mehreren Endzuständen eingeleitet. Im Protokoll werden sodann die möglichen Nachrichtenempfangs- und Nachrichtensendevorgänge bestimmten Zustandsübergängen zugeordnet und somit alle im Sinne einer korrekten und sinnvollen Nutzung der Funktionalität einer Komponente möglichen Aufruf- bzw. Sendereihenfolgen festgelegt. Zustandsübergänge besitzen die Form

$$\langle \texttt{s1} \rangle : \langle \texttt{direction} \rangle \langle \texttt{message} \rangle \mapsto \langle \texttt{s2} \rangle$$

wobei $\langle \texttt{s1} \rangle$ den Ausgangszustand, $\langle \texttt{s2} \rangle$ den erreichten Zustand bezeichnet, $\langle \texttt{direction} \rangle$ die Kommunikationsrichtung angibt (»+« für Empfang, »–« für Senden) und $\langle \texttt{message} \rangle$ der Name einer Nachricht ist, der in der Schnittstellenspezifikation festgelegt wurde. Die Zustandswerte entsprechen nicht notwendigerweise internen Zuständen innerhalb der Komponente. Der Komponentenentwickler kann Zustandsbezeichnungen wählen, die von den konkreten internen Zuständen der Komponente abstrahiert sind.[11]

Mittels des Protokolls wird festgelegt in welchem Zustand welche(r) Kommunikationsschritt(e) möglich ist (sind). Die Protokolle der beteiligten Kommunikationspartner legen damit konkret alle möglichen Ablaufszenarien ihrer Interaktion fest.

[11] Dies gleicht dem Zustandsbegriff, der in Abschnitt 1.1.5 verwendet wurde und erlaubt insbesondere die konsequente Aufrechterhaltung der Black-Box-Sicht.

Die Repräsentation des Protokolls selbst basiert auf dem Konzept endlicher Automaten. Es existiert eine endliche Menge diskreter Zustände, von denen einer als Anfangszustand und beliebig viele als Endzustände gekennzeichnet werden. In einem bestimmten Zustand des Protokolls darf eine Komponente nun Nachrichten empfangen und / oder Nachrichten senden. Ein Zustand, in dem sowohl Nachrichten empfangen als auch gesendet werden dürfen, wird als *gemischter Zustand* bezeichnet.

Jeder Protokollverlauf einer Kollaboration beginnt im Anfangszustand. In diesem (vom internen Anfangszustand der Komponente abstrahierten) Anfangszustand befindet sich ein auf Basis des Protokolls implementierter endlicher Automat, wenn er initialisiert wird.

Wenn im Verlauf einer Kommunikation in diesem Automaten ein Endzustand erreicht wird, kann davon ausgegangen werden, dass die Kollaboration im Hinblick auf diese Komponente erfolgreich verlaufen ist. Als Endzustand sind demnach Zustände deklariert, in denen die Komponente im Sinne des Aufrufers bzw. der relevanten Domänenkonzepte ihre (Teil-) Funktionalität erbracht hat.

Die bisher eingeführten und im weiteren Verlauf dieses Kapitels noch zu erläuternden Begriffe und Verfahren dieses Modells werden zum besseren Verständnis anhand des nachfolgend dargestellten konkreten Beispielszenarios erörtert. Implementiert werden soll dabei ein Auktionsdienst, bestehend aus einem Auktionator, der Waren versteigert, und einem Bieter, welcher Waren ersteigern kann (vgl. a. [Griffel, 1998]). Die Kollaborationsspezifikation des Auktionators stellt sich dann etwa folgendermaßen dar:

```
Collaboration Auctioneer {
  Receive Messages {
    bid (bidderId:id, itemBiddingOn:string,
         amount:real);
    updateAck();
  }
  Send Messages {
    newItem(itemDescr:string, bidderId:id);
    update(highBid:real);
    rejectBid(reason:string);
    acceptBid();
    itemSold();
  }
  Protocol {
    States { A (init), B, C, D, E (final)};
    Transitions {
```

```
      A:  -newItem    -> B;
      B:  -update     -> C;
      B:  -itemSold   -> A;
      B:  +bid        -> D;
      C:  +updateAck  -> B;
      D:  -rejectBid  -> B;
      D:  -acceptBid  -> E;
      E:  -update     -> C;
      E:  -itemSold   -> A; }
  }
}
```

Beispiel 3.1 *Kollaborationsspezifikation eines Auktionators A mit Protokoll P_A*

Die zu diesem deklarativen Teil gehörende Implementierung eines Auktionsdienstes besitzt folgende Eigenschaften, die zugleich als Vorgaben für die jeweiligen Komponentenentwickler betrachtet werden:

- die Syntax der zu sendenden Methoden und ihrer Parameter entspricht exakt den Angaben in der Kollaborationsspezifikation,
- die Syntax der zu empfangenen Methoden und ihrer Paramter entspricht exakt den Angaben in der Kollaborationsspezifikation,
- bei Initialisierung der Komponente befindet sie sich in einem internen Zustand, der dem Anfangszustand des Protokolls assoziiert ist,
- die internen Zustandsveränderungen, die die Komponente durchläuft, können den im Protokoll aufgelisteten Zustandsübergängen entsprechen.

3.3.2 Technische Anforderungen an die Systemumgebung

Eine dieses Kommunikationsmodell optimal implementierende Systemumgebung sollte den Protokollen entsprechende Automaten zum Beginn der Interaktion zweier Komponenten (nach vorheriger Kopplung) generieren und initialisieren (in den Anfangszustand versetzen). Im Verlauf der Kommunikation werden dann die jeweiligen Zuständsübergänge in einzelne Schritte des Automaten umgesetzt. Zudem werden beim Nachrichtenaustausch die Methodennamen und die übergebenen Parameter aufgrund

der Kollaborationsspezifikation auf Korrektheit geprüft. Daher wird in der in Abschnitt 6.1 erläuterten Implementierung jede gesendete und zu empfangende Nachricht über *aktive* Komponenten (Interzeptoren) weitergeleitet. Die Zusammenfassung von je zwei der jeweils genau einer Komponente zugeordneten Interzeptoren stellt dann den die beiden Komponenten verbindenden Konnektor dar (vgl. a. Abbildung 3.5 auf Seite 148).

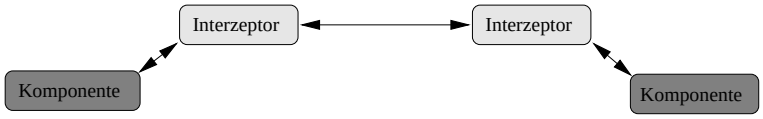


Abbildung 3.1 *Nachrichtenaustausch mittels Interzeptoren*

3.4 Kompatibilitätsprüfung

3.4.1 Konzept

Wenn eine Anwendung durch Komposition verschiedener Komponenten erstellt wird, ist es nun anhand der oben beschriebenen Kollaborationsspezifikation möglich, zur Erstellungszeit die Kompatibilität der beteiligten Komponenten zueinander zu prüfen. Die Kompatibilität ist gegeben, wenn sowohl die Schnittstellenspezifikationen als auch die Protokolle der Kollaborationsspezifikationen zweier potenziell miteinander interagierenden Komponenten zueinander kompatibel sind.[12]

Aus Anwendersicht ist neben diesem Kompatibilitätskriterium zusätzlich von Bedeutung, dass möglichst beide Komponenten bei einem (evtl. vorläufigen) Abschluss ihrer Interaktion sich jeweils gemäß ihrer Protokolle in einem Endzustand befinden – denn dies ist das Kriterium für eine erfolgreiche Diensterbringung einer Komponente[13] (siehe Abschnitt 3.3.1).

[12] Von einer semantischen Kompatibilität kann dann gesprochen werden, wenn sich zudem beide Kollaborationsspezifikationen des standardisierten Vokabulars einer gemeinsamen Anwendungsdomäne bedienen.

[13] Falls seitens des Anwendungsentwicklers keine Endzustände definiert werden, kann möglicherweise das Erreichen von Anfangszuständen im Kommunikationsablauf als erfolgreiche Diensterbringung betrachtet werden.

3.4.2 Kompatibilität der Schnittstellenspezifikationen

Die Kompatibilität zweier Schnittstellenspezifikationen lässt sich durch den Vergleich der jeweils angegebenen Empfangs- und Sendenachrichten miteinander prüfen. Dabei gilt:

Definition 3.3 (Kompatibilität Schnittstellenspezifikation) *Zwei Schnittstellenspezifikationen S_1 und S_2 sind zueinander kompatibel, wenn zu jeder Sendenachricht in S_1 eine Empfangsnachricht in S_2 gleichen Namens (mit Parametern gleichen Namens in gleicher Reihenfolge) und zu jeder Sendenachricht in S_2 eine Empfangsnachricht in S_1 gleichen Namens (mit Parametern gleichen Namens in gleicher Reihenfolge) existiert.*

3.4.3 Kompatibilität der Protokolle

Die Kompatibilitätsprüfung der Protokolle basiert auf der Untersuchung aller möglichen Kommunikationsabläufe ausgehend vom Protokoll der Kollaborationsspezifikation[14]. Die Kompatibilität der Protokolle ist sichergestellt, wenn folgendes gilt:

Definition 3.4 (Protokollkompatibilität) *Zwei Protokolle P_1 und P_2 sind zueinander kompatibel, wenn gilt: Immer wenn P_1 in einem Zustand ist, in der es eine bestimmte Nachricht senden kann, dann ist P_2 in einem Zustand, in dem es diese Nachricht empfangen kann. Gleiches gilt aus Sicht von P_2. Zudem befindet sich in den jeweiligen Anfangszuständen und nach jedem gemeinsamen Kommunikationschritt immer mindestens ein Protokoll in einem Zustand, in dem es eine Nachricht senden kann, oder beide Protokolle befinden sich in einem Endzustand.*

Zur Illustration einer auf Protokollen basierenden Kompatibilitätsprüfung (und zur Motivation der obigen Definition der Protokollkompatibilität) dienen im Folgenden einige Beispiele. Zunächst wird eine zu Beispiel 3.1 auf Seite 134 *kompatible* Kollaborationsspezifikation betrachtet:

```
Collaboration Bidder {
  Receive Messages {
    newItem(itemDescr:string, bidderId:id);
    update(highBid:real);
    rejectBid(reason:string);
```

[14] Eine formale Herleitung von Definition 3.4 zur Komponentenkompatibilität und die Basis des in Abschnitt 3.6 präsentierten Algorithmus zur Kompatibilitätsprüfung findet sich im Anhang B.

```
    acceptBid();
    itemSold();
  }
  Send Messages {
    bid (bidderId:id, itemBiddingOn:string,
         amount:real);
    updateAck();
  }
  Protocol {
    States { 1 (init), 2, 3, 4, 5 (final)};
    Transitions {
      1: +newItem    -> 2;
      2: +update     -> 3;
      2: +itemSold   -> 1;
      2: -bid        -> 4;
      3: -updateAck  -> 2;
      4: +rejectBid  -> 2;
      4: +acceptBid  -> 5;
      5: +update     -> 3;
      5: +itemSold   -> 1;}
  }
}
```

Beispiel 3.2 *Kollaborationsspezifikation eines Bieters B_1 mit Protokoll P_{B1}*

Diese Schnittstellenspezifikation ist zu der in Beispiel 3.1 auf Seite 134 kompatibel, da die Nachrichten und ihre zugehörigen Parameter identische Namen besitzen und die Parameter in identischer Reihenfolge stehen. Zudem entsprechen die Menge der Empfangsnachrichten von A exakt der Menge der Sendenachrichten von B_1 und umgekehrt.

Die Protokolle sind kompatibel, da es immer wenn die zu P_A gehörende Komponente in einem Zustand ist, in dem es eine bestimmte Methode senden kann, die zu P_{B1} gehörende Komponente in einem Zustand ist, in dem es diese Methode empfangen kann – und umgekehrt. Zudem ist zu jedem möglichen Zeitpunkt einer Interaktion dieser Komponenten sichergestellt, dass mindestens eine Komponente eine Nachricht senden kann. Dies ist sichergestellt, da das Protokoll P_{B1} dem Protokoll P_A entspricht – nur die Richtung jeder Nachricht ist genau umgekehrt. Daher wird der Kommunikationsablauf von P_{B1} als zu P_A *komplementär* bezeichnet, notiert als $P_{B1}=\bar{P}_A$. Hieraus folgt:

Definition 3.5 (Kompatibilität Komplementärprotokoll)

Zwei Protokolle P_A und P_B sind zueinander kompatibel, wenn $P_B=\bar{P}_A$.

Dies ist eine hinreichende, jedoch keine notwendige Bedingung für die Kompatibilität von Protokollen. Warum dies so ist, zeigt das folgende Beispiel eines gegenüber Beispiel 3.2 auf der vorherigen Seite leicht modifizierten Protokolls P_{B2}.

```
Collaboration Bidder {
  ...
  Protocol {
    States { 1 (init), 2, 3, 4, 5 (final)};
    Transitions {
      1: +newItem    -> 2;
      2: +update     -> 3;
      2: +itemSold   -> 1;
      2: -bid        -> 4;
      3: -updateAck  -> 2;
      4: +rejectBid  -> 2;
      4: +cancel     -> 1;
      4: +acceptBid  -> 5;
      5: +update     -> 3;
      5: +itemSold   -> 1;}
  }
}
```

Beispiel 3.3 *Kollaborationsspezifikation eines Bieters B_2 mit Protokoll P_{B2}*

Auch P_{B2} ist zu P_A kompatibel, denn ihre korrekte Kommunikation wird durch das erweiterte Protokoll nicht gestört. Komponente A wird nie versuchen eine Methode des Namens `cancel` bei Komponente B_2 aufzurufen, da dies in ihrem eigenen Protokoll nicht vorgesehen ist. Somit wird der entsprechende Zustandsübergang im Protokoll von B_2 in keinem möglichen Kommunikationsablauf dieser beiden Komponenten auftreten.

Damit liegen in der Interaktion zwischen A und B_2 exakt die gleichen möglichen Kommunikationsabläufe vor wie in der Interaktion zwischen A und B_1 und es gilt auch hier, dass die Kollaborationsspezifikationen zueinander kompatibel sind.

Diese Erkenntnis ist im Einklang mit Definition 3.4, die ja lediglich verlangt, dass der Kommunikationspartner eines Protokolls P, das sich in einem Zustand befindet, der das Senden einer Nachricht m erlaubt, auf den möglichen Empfang von m vorbereitet sein muss. Hingegen gibt es keine Notwendigkeit, dass der Kommunikationspartner von P eine Nachricht m senden muss, wenn P den Empfang dieser Nachricht ermöglicht.

3.4.4 Mehrdeutigkeit der Kompatibilitätsprüfung

Definition 3.4 beschreibt die notwendige Bedingung einer Protokollkompatibilität. Diese Bedingung wird beispielsweise durch einen *unspezifizierten Empfang*, der im nächsten Abschnitt betrachtet wird, verletzt. Selbst wenn jedoch die Bedingungen nach Definition 3.4 verletzt werden, ist nicht sicher, dass die Interaktion zweier Komponenten scheitern muss. Das gemeinsame Erreichen von Endzuständen ist ein Anzeichen für Kompatibilität nach dem Kriterium aus Abschnitt 3.4.1. Falls gemeinsam Endzustände erreicht werden *können*, aber auch andere Kommunikationsabläufe existieren, die dieses verhindern würden, lässt dies Spielraum für Interpretationen des Ergebnisses einer derartigen Kompatibilitätsprüfung.

Abschnitt 3.4.7 wird zudem *Endlosschleifen* diskutieren, die zu einem unerwünschten Interaktionsablauf zweier Komponenten führen. Das Vorhandensein einer Endlosschleife verletzt jedoch nicht die Bedingungen nach Definition 3.4. Formal sind zwei Komponenten, bei deren Protokollen in der Interaktion eine Endlosschleife auftritt, zueinander kompatibel.

Die Information, die durch eine Kompatibilitätsprüfung ermittelt wird, ist somit für ein sinnvoll ablaufendes Gesamtsystem nicht ausreichend. In Abschnitt 3.6.1 werden daher mögliche Ergebnisse einer Kompatibilitätsprüfung und ihre Interpretationen näher erörtert.

3.4.5 Unspezifizierter Empfang

Wird nun Protokoll P_{B3} betrachtet, das, da es die Kompatibilitätsbedingung aus Definition 3.4 verletzt, inkompatibel zu P_A ist.

```
Collaboration Bidder {
  ...
  Protocol {
    States { 1 (init), 2, 3, 4, 5 (final)};
    Transitions {
      1: +newItem    -> 2;
      2: +update     -> 3;
      2: +itemSold   -> 1;
      2: -bid        -> 4;
      3: -updateAck  -> 2;
      3: -stop       -> 1;
      4: +rejectBid  -> 2;
      4: +acceptBid  -> 5;
      5: +update     -> 3;
      5: +itemSold   -> 1;}
```

```
  }
}
```

Beispiel 3.4 *Kollaborationsspezifikation eines Bieters B_3 mit Protokoll P_{B3}*

Hier kann nun im Kommunikationsablauf die Situation eintreten, dass Komponente B_3 eine Nachricht `stop` sendet, die im Protokoll von Komponente A nicht vorgesehen ist.

Es ist für die Kompatibilität zweier Protokolle unbedeutend, ob eine Komponente Methoden empfangen darf, die im Protokoll einer mit ihr verbundenen Komponente nicht als zu sendende Methoden vorgesehen sind. Wenn hingegen eine Komponente eine Methode senden darf, deren Empfang in der mit ihr verbundenen Komponente nicht vorgesehen ist, so sind diese Protokolle inkompatibel. Dies wird als *unspezifizierter Empfang* bezeichnet.

3.4.6 Verklemmung (deadlock)

Eine *Verklemmung* tritt auf, wenn es im Kommunikationsverlauf zwischen den beteiligten Komponenten zu einer Situation kommt, in der keine der beiden Komponenten sich in einem Zustand befindet, der das Senden einer Nachricht erlaubt, die der Kommunikationspartner in seinem Zustand empfangen kann. Es existiert dann für beide Komponenten keine Möglichkeit eines Zustandsüberganges – also keine Möglichkeit zur Fortsetzung der Kommunikation. Wenn in einem derartigen Zustand eine Komponente etwas senden könnte, so müsste sie es nach den in Abschnitt 3.5 erläuterten Voraussetzungen auch tun – und es wäre an dieser Stelle keine Verklemmung aufgetreten.

Wenn der Zustand, in dem die Verklemmung auftritt, ein deklarierter Endzustand für die jeweilige Komponente ist, dann ist das Auftreten dieser Verklemmung nicht als Hinweis einer Inkompatibilität zu interpretieren. Das Erreichen eines Endzustandes bedeutet, dass eine Komponente die jeweilige (Teil-) Funktion erfolgreich beendet hat. Wenn dieser Zustand nun nicht wieder verlassen werden kann, so ist dies entweder vom Komponentenentwickler vorgesehen oder ein Fehler, der in der Interaktion auftritt[15]. Die Entscheidung darüber kann nur unter Hinzunahme semantischer Informationen über angebotene Dienste und Wissen über die Ziele der Interaktion getroffen werden. Der

[15] falls vom Komponentenentwickler eine Fortsetzung der Interaktion nach Erreichen eines Endzustandes vorgesehen war

hier vorgestellte Algorithmus kann das nicht entscheiden.[16] Dies ist zugleich ein weiterer Beleg für die in Abschnitt 3.4.4 bereits erwähnte Mehrdeutigkeit der Kompatibilitätsprüfung.

3.4.7 Endlosschleife

Eine *Schleife* tritt in der Kommunikation zweier Komponenten auf, wenn ein aus den jeweils aktuellen Zuständen (basierend auf den zugehörigen Protokollen) zusammengesetztes Zustandspaar im Verlauf einer Kommunikationsabfolge mehr als einmal auftritt. Einen derartigen Kommunikationsablauf illustriert Abbildung 3.2 ausgehend von den Kollaborationsspezifikationen aus den Beispielen 3.1 auf Seite 134 und 3.2 auf Seite 137. Unerwünscht aus Sicht des Anwendungsentwicklers sind *Endlosschleifen*. Damit wird eine sich unendlich wiederholende Kommunikationsabfolge bezeichnet, in der sich Zustandspaare in immer identischer Abfolge wiederholen. Falls in dieser Abfolge keine Endzustände enthalten sind, kann der Anwender für dieses Interaktion zumindest nicht den Zeitpunkt des Abschlusses einer erfolgreichen Diensterbringung bestimmen.

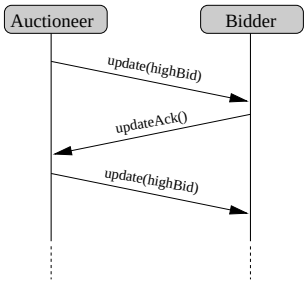


Abbildung 3.2
Das Auftreten einer (potenziell endlosen) Schleife im Kommunikationsablauf

Es ist in dem hier vorliegenden Modell nicht generell entscheidbar, ob eine Schleife verlassen wird oder nicht. Wenn es im Verlauf der Schleife für eine oder beide Komponenten die Möglichkeit auftritt, die Schleife zu verlassen[17], dann ist es unentscheidbar, ob diese Möglichkeit genutzt wird oder nicht. Dies liegt darin begründet, dass anhand des Protokolls nicht ersichtlich ist, welche Nachricht in einem solchen Zustand zum Senden ausge-

[16] Daher bedarf er zur sinnvollen Anwendung der Einbettung in eine domänenspezifische Klassifikation wie sie im folgenden Kapitel vorgestellt wird.

[17] Die ist potenziell in Zuständen gegeben, in denen *unterschiedliche* Nachrichten gesendet werden können.

wählt wird – das entscheidet entscheidet sich in der internen Implementierung der Komponente. Zudem ist es in gemischten Zuständen anhand des Modells unmöglich zeitliche Abläufe vorherzusehen – also welche Komponente den nächsten Sendevorgang einleiten wird.

Das Auftreten von (potenziell endlosen) Schleifen lässt sich durch den in Abschnitt 3.6.2 vorgestellten Algorithmus der Kompatibilitätsprüfung nicht ermitteln[18]. Schleifen sind im Rahmen des vorliegenden Modells lediglich von peripherem Interesse, da – wie oben erläutert – der Nachweis einer Endlosschleife unmöglich ist und Schleifen, die in endlicher Zeit verlassen werden, für die Kompatibilität interagierender Komponenten unbedeutend sind.

3.5 Laufzeitsynchronisation

Es werden für eine erfolgreiche Anwendung des beschriebenen Kommunikationsmodells einige wichtige Bedingungen zum (Laufzeit-) Verhalten der beteiligten Komponenten und ihrer Interaktion festgelegt:

- Eine Komponente wird nicht unendlich in einem Zustand verharren, in dem sie etwas senden kann.
- Die Kommunikation ist synchronisiert (dies wird hier auch als *lock step* bezeichnet). wechselseitige Methodenaufrufe erfolgen nacheinander – nicht gleichzeitig.
- Eine Komponente verarbeitet maximal eine Nachricht zur Zeit.[19]

Bei nicht synchronisiertem Kommunikationsblauf würden beispielsweise bei der Kommunikation zweier Komponenten, deren Protokolle P_A und P_{B1} aus den Beispielen 3.1 auf Seite 134 und 3.2 auf Seite 137 entnommen sind, Situationen des unspezifiziertem Empfangs bzw. deadlock auftreten können.

Die Ursache liegt dabei in den gemischten Zuständen der beiden Protokolle. Angenommen eine Komponente AUK implementiert P_A und eine Komponente BID implementiert P_{B1}. Wenn

[18] In Anhang B wird gezeigt, wie ein Algorithmus zum Auffinden von Schleifen im Kommunikationsablauf zweier Komponenten konstruiert werden kann.

[19] Insofern stellt ein Methodenaufruf auch im nebenläufigen Fall eine atomare Nachricht im Sinne des Komponentenmodells dar.

AUK im Zustand »B« eine Nachricht `update` schickt (und in Zustand »C« übergeht) und BID (die sich zu diesem Zeitpunkt in Zustand »2« befindet) eine Nachricht `bid` sendet (und in Zustand »4« übergeht) *bevor* sie die Nachricht `update` erhält, kommt es auf beiden Seiten zu einem unspezifizierten Empfang – und als Folge zur Verklemmung.

Eine Synchronisation der Kommunikation lässt sich mittels eines Konnektors erreichen, der dafür sorgt, dass eine Komponente immer erst dann eine Nachricht senden darf, wenn die letzte gesendete Nachricht korrekt empfangen wurde.[20] Konkret sind zwei Umsetzungen denkbar:

- Eine Komponente muss vor dem Senden einer Nachricht beim Konnektor eine entsprechende Bestätigung einholen.

- Es darf zu beliebigen Zeitpunkten gesendet werden, aber jede Komponente muss sich darauf einstellen, dasss ihre Nachricht vom Konnektor zurückgewiesen wird. Dann müssen von den Komponenten Mechanismen implementiert sein, die ein Wiederherstellen des Zustandes vor dem Senden der Nachricht ermöglichen (rollback). Gleiches gilt für einen auf Basis des zugehörigen Protokolls implementierten Automaten.

Ein derartiger Konnektor muss in beiden Fällen eine übergeordnete Instanz darstellen, welche die Kommunikation steuert. Implementiert werden kann der Konnektor als eigene Komponente. Das Schema aus Abbildung 3.1 wird somit um eine konzeptuell *zwischen* den Interzeptor-Komponenten sitzende Konnektor-Komponente erweitert.[21] Die Aufgaben dieser Komponente können auch durch eine Adapter-Komponente, deren eigentliche Funktionalität in Abschnitt 3.7 erläutert wird, erfüllt werden – es soll dann von einem *adaptierenden Konnektor* gesprochen werden. Abbildung 3.5 illustriert diese Konzeption.

[20] Die aktive Rolle eines Konnektors erlaubt das Abfangen »verfrühter« Nachrichten.

[21] Implementatorisch kann die Gesamtfunktionalität der an der Verbindung beteiligten Bauelemente sehr wohl in einer einzigen Komponente realisiert sein.

3.6 Algorithmus zur Kompatibilitätsprüfung

Im Folgenden soll der im Rahmen dieser Arbeit implementierte Algorithmus vorgestellt und erörtert werden, der eine Kompatibilitätsprüfung zweier Protokolle ermöglicht.

Basis dieses Algorithmus sind die in Abschnitt 3.1.1 vorgestellten Kriterien syntaktischer Komponentenkompatibilität (insbesondere Definition 3.4) und die in Anhang B präsentierten formalen Definitionen. Das Ergebnis dieser Prüfung fällt differenzierter aus, als eine binäre »ist kompatibel / ist nicht kompatibel«-Antwort. Dies ist auch erwünscht, da Protokolle möglich sind, bei denen eine solche undifferenzierte Aussage nicht angebracht ist (vgl. Abschnitt 3.4.4).

3.6.1 Rückgabeinformationen

Folgende Informationen werden vom entsprechenden Algorithmus geliefert:

- eine Kommunikation findet statt / findet nicht statt,
- die Endzustände werden erreicht / werden nicht erreicht,
- die Anfangszustände werden (erneut) erreicht / werden nicht erreicht,
- eine Verklemmung kann auftreten / kann nicht auftreten, (deadlock eines Endzustand wird dabei nicht gewertet, s.o.),
- ein unspezifizierter Empfang kann auftreten / kann nicht auftreten.

Mittels dieser Informationen kann ein Anwendungsentwickler oder ein Werkzeug eine Bewertung der syntaktischen Kompatibilität zweier Komponenten vornehmen.[22]

Das optimale Ergebnis einer Kompatibilitätsprüfung ist erreicht, wenn eine Kommunikation stattfindet, die Endzustände erreicht werden, keine Verklemmung und kein unspezifizierter Empfang auftritt. Abschnitt 3.4.4 hat aber auch gezeigt, dass Ergebnisse, die vom Optimalfall abweichen, durchaus tolerabel sein können.

[22] Die Information »Kommunikation findet statt / findet nicht statt« wird benötigt, da Komponentenprotokolle eine Inkompatibilität zueinander aufweisen können, die ausgehend vom jeweiligen Anfangszustand keinerlei Kommunikationsschritte zulässt (vgl. a. Definition 3.4).

3.6.2 Ablauf des Prüfungsalgorithmus

Betrachtet werden Zustandspaare $(\mathtt{s}, \mathtt{t})$, wobei s und t die Zustände der Protokolle der beteiligten Komponenten (als P_A und P_B bezeichnet) zu einem diskreten Zeitpunkt sind. Von einem Zustandspaar $(\mathtt{s}, \mathtt{t})$ lässt sich ein Nachfolgezustandspaar $(\mathtt{s}', \mathtt{t}')$ herleiten, wenn eine Nachricht m existiert, sodass entweder

- $\mathtt{s} : -\mathtt{m} \mapsto \mathtt{s}' \in \mathtt{P_A}$ und $\mathtt{t} : +\mathtt{m} \mapsto \mathtt{t}' \in \mathtt{P_B}$ oder
- $\mathtt{t} : -\mathtt{m} \mapsto \mathtt{t}' \in \mathtt{P_B}$ und $\mathtt{s} : +\mathtt{m} \mapsto \mathtt{s}' \in \mathtt{P_A}$

Der Algorithmus beginnt mit der Bildung der Mengen EQUIV, NEW und WORK. In EQUIV und NEW befindet sich zunächst jeweils das Zustandspaar, das aus den beiden Anfangszuständen der beteiligten Protokolle gebildet wird. WORK beinhaltet zunächst die leere Menge. Mit dem einzigen Element der Menge NEW wird nun ein erster Durchgang gestartet.

Es werden alle Nachrichten, die die beteiligten Komponenten in ihren Protokollen vorsehen, daraufhin geprüft, ob sich mittels eines Sendevorganges der jeweiligen Nachricht ein Zustandsübergang herbeiführen lässt. Falls eine Prüfung erfolgreich ist, wird das Protokoll des potenziellen Kommunikationspartners auf einen Zustandsübergang durch Empfang dieser Nachricht getestet. Wenn ein derartiger Übergang nicht existiert, wird ein unspezifizierter Empfang vermerkt. Andernfalls wird ein Zustandspaar aus den beiden Zuständen, in denen sich die Protokolle nach den Zustandsübergängen befinden, gebildet und der Menge WORK hinzugefügt. Falls bei der Prüfung eines Zustandspaares kein Nachfolgezustandspaar hergeleitet werden kann, so wird eine Verklemmung vermerkt.

Der Menge NEW werden im Laufe dieser Untersuchung nacheinander alle Elemente entnommen (vgl. Abbildung 3.4). Ist ein Durchgang unter Anwendung aller potenziell sendbaren Nachrichten auf beide Protokolle abgeschlossen, wird die Menge WORK analysiert.

- Enthält WORK lediglich die leere Menge, so wird der Algorithmus mit Ausgabe des Resultats beendet.
- Enthält WORK das Anfangs- oder ein Endzustandspaar, so wird das Erreichen dieses Zustandspaares vermerkt.
- Elemente von WORK, die sich nicht in EQUIV befinden, werden sowohl der Menge NEW als auch der Menge EQUIV hinzugefügt

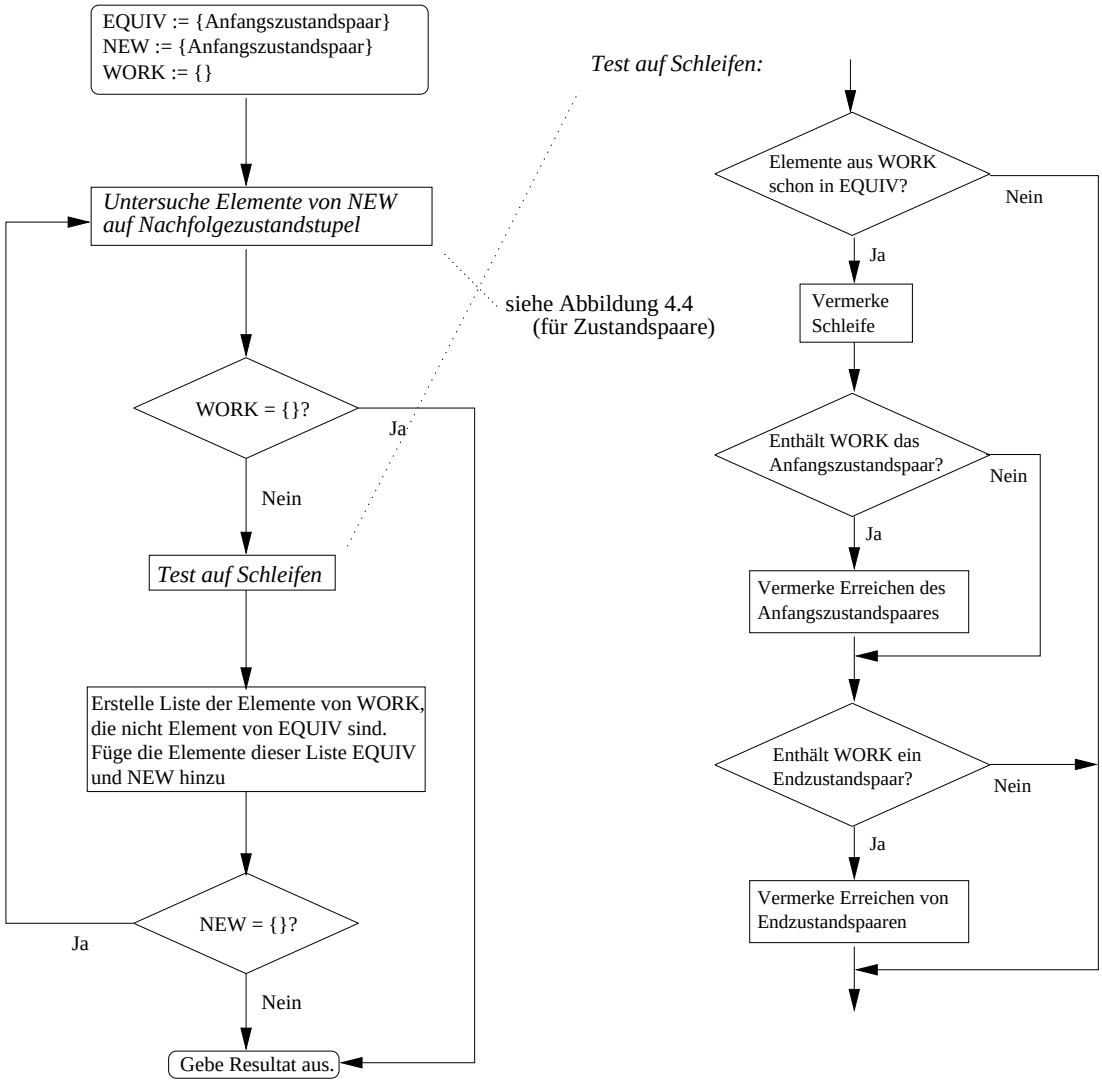


Abbildung 3.3
Ablaufdiagramm des Algorithmus zur Kompatibilitätsprüfung

Im Anschluss wird die Menge `NEW` betrachtet: Enthält `NEW` lediglich die leere Menge, so wird der Algorithmus beendet. Andernfalls werden nun weitere Durchgänge analog dem oben beschriebenen Verfahren mit den einzelnen Elementen der Menge `NEW` durchgeführt.

Der Algorithmus wird beendet, sobald `NEW` nach einem kompletten Durchgang die leere Menge enthält. Dem Softwarekonstrukteur können nun die oben genannten Informationen (vgl. a. Abschnitt 3.6.1 und Kapitel 4) ausgegeben werden. Der vollstän-

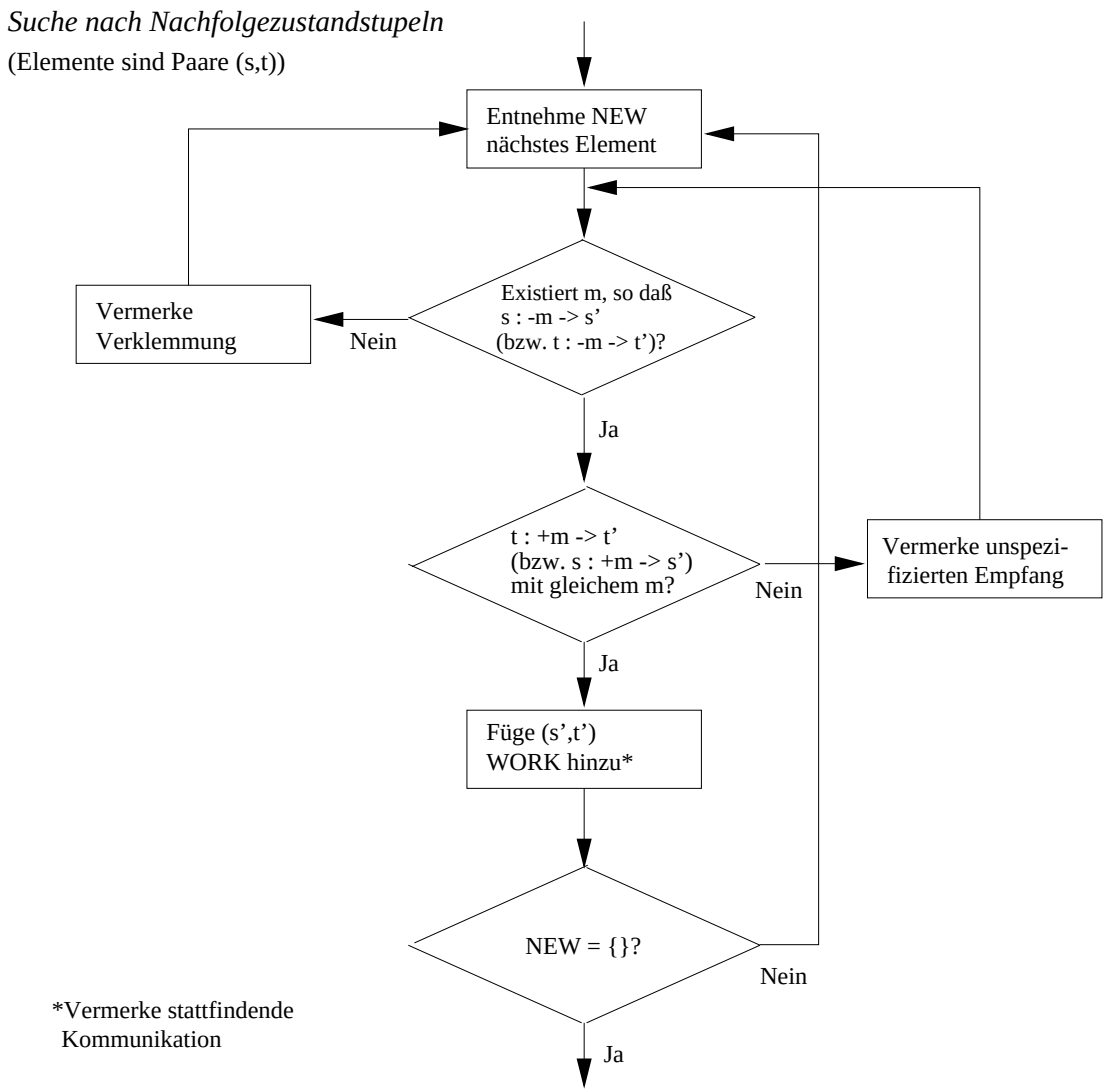


Abbildung 3.4
Ablaufdiagramm des Algorithmus zur Suche nach Nachfolgezustandspaaren

dige Algorithmus wird in den Ablaufdiagrammen in den Abbildungen 3.3 und 3.4 graphisch dargestellt.

3.7 Adapter

3.7.1 Motivation

Zur Ermöglichung der Kommunikation zwischen zwei Komponenten, deren Implementierungen anhand ihrer Kollaborations-

spezifikationen nicht kompatibel sind, lassen sich Adapter einsetzen. Der Adapter sitzt als Konnektor zwischen zwei kommunizierenden Komponenten und erhält sämtliche Nachrichten, die für den jeweiligen Interaktionspartner gedacht sind. Der Adapter selbst ist die einzige Komponente in diesem Modell, die Nachrichten an die beteiligten (nicht kompatiblen) Komponenten über entsprechende Interzeptoren senden darf.

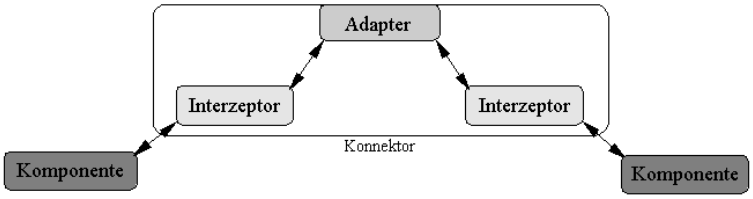


Abbildung 3.5 *Nachrichtenaustausch mittels Interzeptoren über eine Adapterkomponente*

Es soll nun eine Bieter-Komponente betrachtet werden, die von der in Beispiel 3.2 auf Seite 137 grundlegend abweicht. Denkbar wäre, dass ein Entwickler eine Bieter-Komponente mit folgender Kollaborationsspezifikation entwirft:

```
Collaboration Bidder {
  Receive Messages {
    auctionBegin(auctionInfo:string);
    canBid(bidId:id);
    cannotBid(why:string);
    newHighBid(price:real);
    bidOk();
    bidNotOk();
    gavel();
  }
  Send Messages {
    requestTobid (name:string, bidItem:string);
    newBid(bidId:id, myBid:real);
    highBidAck();
  }
  Protocol {
    States { 1 (init), 2, 3, 4, 5, 6, 7 (final)};
    Transitions {
      1: +auctionBegin -> 2;
      2: -reqToBid     -> 3;
      3: +cannotBid    -> 1;
      3: +canBid       -> 4;
      4: +newHighBid   -> 5;
      4: +gavel        -> 1;
      4: -newBid       -> 6;
```

```
      5: -highBidAck   -> 4;
      6: +bidNotOk     -> 4;
      6: +bidOk        -> 7;
      7: +newHighBid   -> 5;
      7: +gavel        -> 1;}
  }
}
```

Beispiel 3.5 *Kollaborationsspezifikation eines Bieters B_4 mit Protokoll P_{B4}*

Die Kommunikation mit dem Auktionator nach der Kollaborationsspezifikation aus Beispiel 3.1 auf Seite 134 ist für einen derartigen Bieter aus folgenden Gründen zunächst einmal unmöglich:

- Die Methodennamen sowohl der zu sendenden als auch der empfangbaren Nachrichten sind unterschiedlich.
- Die Namen der beteiligten Parameter unterscheiden sich ebenfalls.

Diese rein syntaktischen Abweichungen ließen sich durch einen einfachen Adapter, der lediglich als eine Art Übersetzer fungiert, ausgleichen, falls die unterschiedlich benannten Methodenaufrufe semantisch übereinstimmen (funktional gleichartige Bedeutung tragen).

Ein solcher Adapter bräuchte nur eine interne Tabelle,[23] die die Abbildung verschiedener Methoden- und Parameternamen aufeinander (semantisch sinnvoll) spezifiziert. Somit wäre er in der Lage, von einer Komponente eingehende Nachrichten an den Kopplungspartner so weiterzuleiten, dass diese den in seiner Kollaborationsspezifikation festgelegten Methoden- und Parameternamen entsprechen.

Die Beziehung zwischen dem Auktionator (nach Beispiel 3.1 auf Seite 134) und dem Bieter (nach Beispiel 3.5) ist allerdings nicht derart trivial. Es existieren hier zusätzlich zu den oben beschriebenen noch weitere Inkompatibilitäten im Kommunikationsablauf.

Zunächst einmal stimmt auch die Anzahl der Parameter von Methodenaufrufen mit gleicher semantischer Bedeutung nicht immer überein (Beispiel: `+bid` beim Auktionator und `-newBid`

[23] Hier wären auch beliebig komplexe Verfahren zur Wissensrepräsentation denkbar, die jedoch in der Regel die Praktikabilität des Ansatzes untergraben.

beim Bieter). Ursache hierfür ist, dass die Bieter-Komponente offensichtlich Parameter erwartet und verarbeitet, die der Auktionator gar nicht vorgesehen hat. Zu solchen Parametern lässt sich somit auf Seiten des Auktionators kein passendes Gegenstück finden.

Zudem weicht das Protokoll des Bieters von dem des Auktionators ab. Während beim Auktionator-Protokoll lediglich fünf Zustände auftreten, sind es beim Bieter-Protokoll sieben. Diese Differenz ergibt dadurch, dass der Bieter vor der eigentlichen Teilnahme an der Auktion eine Erlaubnisanfrage (`reqToBid`) stellt und erst nach Erhalt einer positiven Antwort (`bidAllowed`) in einen Zustand übergeht, in dem er ein Gebot abgeben kann. Für die Zustände mit den Bezeichnungen »2« und »3« im Protokoll des Bieters gibt es somit keine Entsprechung im Protokoll des Auktionators.

Ein korrektes Modell für mögliche Adapter darf daher nicht nur einfache »Übersetzer« vorsehen, sondern muss in der Lage sein, auch die zuletzt beschriebenen Inkompatibilitäten im Kommunikationsablauf auszugleichen.

Konkret bedeutet dies, dass der Adapter neben der Fähigkeit zur variablen Kommunikations*steuerung* auch die Möglichkeit zum Speichern von im Kommunikationsablauf potenziell noch benötigten Parameterwerten und zum Auslesen dieser gespeicherten Werte besitzen muss.

3.7.2 Konzeption des Adapters

Im Folgenden soll nun ein Adapterkonzept vorgestellt werden, das die Erstellung korrekter Adapter im Hinblick auf die oben genannten Bedingungen ermöglicht. Anhand des schon vorgestellten Auktionator-Bieter Beispiels wird zudem ein zwischen diesen Komponenten korrekt vermittelnder Adapter vorgestellt.

Das Konzept sieht vor, dass ein Adapter einen endlichen (Protokoll-) Automaten implementiert und Schnittstellen zu den Komponenten besitzt, die im Rahmen einer Komposition interagieren sollen. Wenn eine Komponente eine Nachricht sendet, so wird diese vom Adapter abgefangen (Interzeption) und gemäß seines eigenen Protokolls interpretiert. Dieses Protokoll ähnelt dabei dem Protokoll der Kollaborationsspezifikation einer generellen Komponente (vgl. Abschnitt 3.3.1), sodass insgesamt ein sehr homogenes Modell entsteht. Der Adapter kann somit durch den Empfang einer Nachricht (die eigentlich für eine der beteiligten Komponenten bestimmt war) oder durch das Senden einer Nachricht in einen neuen Zustand gelangen.

Es existieren im Protokoll Zustände, in denen der Adapter eine oder mehrere Nachrichten auf diese Weise empfangen kann und es gibt Zustände, in denen der Adapter genau eine Nachricht senden kann und wird (vgl. Abschnitt 3.7.6).

In Erweiterung der Protokollsyntax für Kollaborationsspezifikationen kann bei jedem Zustandsübergang eine *Aktion* angegeben werden, die der Adapter beim Wechseln des Zustandes (also beim Empfang oder dem Senden einer Nachricht) ausführen muss. Solche Aktionen umfassen insbesondere die Speicherung, das Löschen und das Laden von Parametern.

3.7.3 Formale Beschreibung

Formal besteht ein Adapter aus einer endlichen Menge an *Zuständen*, *Speicherzellen* und *Regeln*.

Ein Zustand wird in diesem Modell Tupel der Art `(s1, s2, memCells)` als Werte annehmen, wobei `s1` und `s2` die Zustände der Protokolle der beteiligten Komponenten zu dem Zeitpunkt angeben, in dem der Adapter sich in diesem Zustand befindet, und `memCells` eine Auflistung der zu diesem Zeitpunkt gültigen Speicherzellen, d.h. der Speicherzellen mit geladenen Werten, darstellt. Die Speicherzellen sind in der Lage die Werte aller während der Kommunikation möglicherweise auftretenden Parameter zu speichern.

Regeln können eine der folgenden beiden Formen annehmen:

- $\langle \texttt{s1} \rangle : + \langle \texttt{message} \rangle \texttt{from} \langle \texttt{rolename} \rangle \mapsto \langle \texttt{s2} \rangle [, (\texttt{save} - \texttt{actions})]$ $[, (\texttt{invalidate} - \texttt{actions})]$;
- $\langle \texttt{s1} \rangle : - \langle \texttt{message} \rangle \texttt{from} \langle \texttt{rolename} \rangle \mapsto$ $\langle \texttt{s2} \rangle [, (\texttt{synthesis} - \texttt{actions})]$ $[, (\texttt{invalidate} - \texttt{actions})]$;

Die erste Regel besagt, dass ein Adapter, der sich im Protokollzustand $\langle \texttt{s1} \rangle$ befindet, durch Empfang einer Nachricht $\langle \texttt{message} \rangle$ von einer Komponente, die $\langle \texttt{rolename} \rangle$ als Bezeichner besitzt, in den Zustand $\langle \texttt{s2} \rangle$ übergeht. Dabei können optional $(\texttt{save} - \texttt{actions})$, die in der Form `write(x1,x2„...)` angegeben werden und die Parameter auflisten, die bei dieser Regel in den Speicherzellen gespeichert werden, spezifiziert werden. Die entsprechenden Speicherzellen werden somit *gültig*. Desweiteren besteht die Möglichkeit zur Angabe von $(\texttt{invalidate} - \texttt{actions})$. Diese listen zu löschende Parameter in der Form `invalidate(x1,x2.,...)` auf, welche damit ihre Gültigkeit verlieren.

Die zweite Regel besagt, dass ein Adapter, der sich im Protokollzustand $\langle \texttt{s1} \rangle$ befindet, durch das Senden einer Nach-

richt ⟨message⟩ an eine Komponente, die ⟨rolename⟩ als Bezeichner besitzt, in den Zustand ⟨s2⟩ übergeht. Dabei *muss* spezifiziert werden, wo die Werte der einzelnen Parameter der Nachricht ⟨message⟩ ausgelesen werden. Die (synthesis – actions), die in der Form `param=x` oder `param=f(x1,x2,...)`[24] (wobei `x,x1,x2...` entweder Konstanten oder Speicherzellen sind) angegeben werden, dienen diesem Zweck. Auch hier lassen sich (invalidate – actions) spezifizieren, deren Form und Bedeutung den (invalidate – actions) denen der ersten Regel identisch ist.

3.7.4 Anwendung des formalen Konzepts

Ein nach diesen Regeln implementierter Adapter ist in der Lage, die in den Abschnitten 3.1.2.3ff. formulierten Anforderungen zu erfüllen. Neben einer einfachen Übersetzungsmethodik zur korrekten Weiterleitung mittels Änderung der Bezeichner für Methoden- und Parameternamen, erlaubt ein nach dieser Syntax mit der entsprechenden Funktionalität implementierter Adapter das zwischenzeitliche Speichern von Parameterwerten.

Zu beachten ist, dass ein funktionaler Unterschied zwischen den Regeln des Protokolls einer Kollaborationsspezifikation und den Regeln eines *Adapterprotokolls* besteht. Die Regeln in einer Kollaborationsspezifikation sind eine Beschreibung des (nach aussen sichtbaren) Kommunikationsverhaltens einer Komponente, das durch das interne Verhalten der Komponente (somit durch seine Implementierung) festgelegt ist. Hingegen legen die Regeln eines Adapterprotokolls das Kommunikationsverhalten des Adapters fest. Die Implementierung eines Adapters besteht somit aus einer Auswertung und Umsetzung (in externes Kommunikationsverhalten) dieser Regeln.

Daraus resultiert ein Entscheidungsproblem innerhalb eines Adapters, wenn Zustände vorliegen, in denen die Regeln neben dem Senden einer Nachricht an eine Komponente noch das Senden oder Empfangen weiterer Nachrichten erlauben. Es existiert keine Instanz, die entscheiden würde, wie der Adapter in einem solchen Zustand reagieren soll. Sein Verhalten an dieser Stelle ist *nicht-deterministisch*. Adapter, die bei Zustandstupeln, die das Senden einer Nachricht seitens des Adapters erlauben, keine weiteren Zustandsübergänge vorsehen, werden hingegen als *deterministisch* bezeichnet.

[24] Dies ermöglicht die Anwendung einer vorab definierten Funktion auf den auszulesenden Wert.

Der Adapterimplementierung, die das Adapterprotokoll verarbeitet, wird bei nicht-deterministischen Protokollen ein zufälliges Verhalten erlaubt – die Korrektheit des Adapters bleibt hiervon unberührt.

Bei einer gewöhnlichen Komponente hingegen wäre das Verhalten in einem solchen Zustand durch die interne Implementierung vorgegeben. Von aussen betrachtet ist aber auch dieses Verhalten zufällig, da in der Kollaborationsspezifikation keine Informationen über die interne Implementierung der Komponente vorliegt.

3.7.5 Praktische Umsetzung

Illustriert wird der Einsatz eines Adapters nun anhand der oben beschriebenen Auktionator- (Beispiel 3.1 auf Seite 134) und Bieterkollaborationsspezifikation (Beispiel 3.5 auf Seite 149). Folgende Regeln beschreiben einen mit diesen beiden Komponenten korrekt zusammenarbeitenden Adapter:[25]

```
<A,1>:   +newItem from auctioneer    -> <B,1>,
         write(itemDescr, bidderId);
<B,1>:   -auctionBegin to bidder     -> <B,2>,
         auctionInfo = Cells.itemDescr,
         invalidate(itemDescr);
<B,2>:   +reqToBid from bidder       -> <B,3>,
         write(bidItem);
<B,2>:   +update from auctioneer     -> <C,2>,
         write(highBid);
<B,3>:   -canBid to bidder           -> <B,4>,
         bidId = Cells.bidderId,
         invalidate(bidderId);
<B,4>:   +update from auctioneer     -> <C,4>;
         write(bidId, myBid);
<B,4>:   +newBid from bidder         -> <B,6>,
         write(bidId, myBid);
<C,4>:   -newHighBid to bidder       -> <C,5>,
         price = Cells.highBid
         invalidate(Cells.highBid);
<C,7>:   -newHighBid to bidder       -> <C,5>,
         price = Cells.highBid
         invalidate(Cells.highBid);
<C,5>:   +highBidAck from bidder     -> <C',4'>;
<C',4'>: +updateAck to auctioneer    -> <B,4>;
```

[25] Aus Gründen der Übersichtlichkeit wurden Regeln in Zusammenhang mit dem Empfang der Nachricht `itemSold` nicht dargestellt (vgl. aber Abschnitt 3.9.4).

```
<B,6>:   -bid to auctioneer          -> <D,6>,
         bidderId = Cells.bidId,
         itemBiddingOn = Cells.bidItem,
         amount = Cells.myBid,
         invalidate(Cells.bidId,Cells.myBid);
<D,6>:   +rejectBid from auctioneer  -> <B',6'>;
<D,6>:   +acceptBid from auctioneer  -> <E,6>;
<B',6'>: -bidNotOk to bidder         -> <B,4>;
<E,6>:   -bidOk to bidder            -> <E,7>;
<E,7>:   +update from auctioneer     -> <C,7>,
         write(highBid);
<C,2>:   -updateAck to auctioneer    -> <B',2'>;
<B',2'>: +update from auctioneer     -> <C,2>,
         write(highBid);
<B',2'>: +reqToBid from bidder       -> <B',3'>,
         write(bidItem);
<B',3'>: -canBid to bidder           -> <B',4'>,
         bidId = Cells.bidderId,
         invalidate(bidderId);
<B',3'>: -newHighBid to bidder       -> <B,5>,
         price = Cells.highBid,
         invalidate(Cells.highBid);
<B,5>:   +highBidAck from bidder     -> <B,4>;
```

Beispiel 3.6 *Regeln eines Adapters*

Jeder Zustand des Adapters ist hierbei durch ein Paar `(s,t)` repräsentiert, wobei `s` und `t` die Zustände der beteiligten Komponentenprotokolle darstellen. Dabei ist zu beachten, dass ein Adapterzustand eigentlich durch ein Tripel `(s,t,memCells)` repräsentiert werden sollte, wobei `memCells` eine Auflistung der in diesem Zustand gültigen Speicherzellen enthält. Im Beispiel wurden Zustände, die sich lediglich durch ihre gültigen Speicherzellen unterscheiden, durch die Bezeichner `(s,t)` bzw. `(s',t')` differenziert.

Zu beachten ist ferner, dass in der Praxis eine eindeutigere Bezeichnung der Parameter wünschenswert ist. Sinnvollerweise könnten die Namen der zugehörigen Nachricht und der Komponente, von der diese Nachricht empfangen wurde, in diesen Parameterbezeicher mit einfließen.

Ein Adapter, der auf diesen Regeln basiert, kann, wenn er als Interzeptor zwischen Auktionator und Bieter Nachrichten abfängt und gegebenenfalls Nachrichten sendet, für einen korrekten Ablauf der Kollaboration sorgen.

Einander äquivalente Methodenaufrufe und Zustände werden mit den jeweils korrekt »übersetzten« Bezeichnern weitergeleitet. Das Problem, dass die Kollaborationsspezifikation des

Bieters aufgrund seiner `reqToBid`-Anfrage inklusive des Wartens auf die Antwort in Zustände gelangt, zu denen keine äquivalenten Zustände auf Seiten des Auktionators existieren, wird gelöst, indem der Adapter selbst nach Empfang von `reqToBid` die erwartete Bestätigung `canBid` an den Bieter sendet und die mittels `reqToBid` übergebenen Parameter speichert. Der für die Bestätigung `canBid` notwendige Parameter wird dabei aus dem zuvor in `auctionBegin` übergebenen Parameter geladen.

An diesem Beispiel werden die flexiblen Möglichkeiten dieses Adapterkonzeptes deutlich. Einander äquivalente Methodenaufrufe und Parameter werden transformiert und gegebenenfalls wird die Weiterleitung verzögert bis sämtliche benötigten Parameter in den Speicherzellen vorliegen. Dies ermöglicht einem Entwickler, die abstrakte Beschreibung eines korrekten Adapters zu formulieren, wobei er nach Möglichkeit auf entsprechende Werkzeuge zurückgreifen können sollte. In Kapitel 6 wird die entsprechende Implementation eines Protokolleditors sowie die Implementierung des Adapterkonzeptes, also die Erzeugung einer auf diesen Regeln basierende Adapterkomponente, die als Konnektor mit den beteiligten Komponenten kommuniziert, vorgestellt.

3.7.6 Laufzeitverhalten

Im Kommunikationsablauf der beteiligten Komponenten nach Abbildung 3.5 sind als Voraussetzungen zu beachten, dass sämtliche von den beiden Komponenten gesendeten Nachrichten ausschließlich an den Adapter geleitet werden und Nachrichten, die der Adapter sendet, jeweils nur an eine der beiden Komponenten gesendet werden.

Zudem übernimmt der Adapter die Steuerung der Kommunikation. Er ist damit in der Lage, eingehende Nachrichten zurückzuweisen. Dies ist beispielsweise nötig, wenn der Adapter den Zustand, in dem er eine bestimmte Nachricht akzeptiert, bereits verlassen hat, da eine andere Regel, die aus diesem Zustand herausführt, bereits angewandt wurde – entweder durch das Senden einer Nachricht oder durch den Empfang einer Nachricht der anderen Komponente.

Im Beispiel 3.6 auf der vorherigen Seite illustriert der Zustand `(B,2)` einen solchen Vorgang: Je nachdem, welche Nachricht der Adapter zuerst erhält, wird er eine der beiden Regeln anwenden und den Zustand verlassen.

Durch derartige Kommunikationssteurung seitens des Adapters werden Inkonsistenzen im Laufzeitverhalten vermieden.

3.7.7 Kritische Anmerkungen

Ein nach den Kriterien aus Abschnitt 3.7.3 erstellter Adapter garantiert nicht die funktional sinnvolle Zusammenarbeit einer Komponentenkomposition mit diesem Adapter.

Neben dem schon in Abschnitt 3.4.7 besprochenen Problem der Endlosschleifen, die in einer Kollaboration mit einem Adapter ebenso auftreten können wie ohne ihn, kann insbesondere »aktives« Kommunikationsverhalten des Adapters als kritisch betrachtet werden.

Aktives Kommunikationsverhalten liegt beispielsweise vor, wenn der Adapter, der ja als kommunikationssteuernder Konnektor in diesem Modell auftritt, eine Nachricht an eine Komponente sendet, von der er zuletzt eine Nachricht erhalten hat, ohne die andere Komponente, die ja der eigentliche Adressat der letzten Nachricht war, mittels Senden einer Nachricht zu benachrichtigen.

In Beispiel 3.6 auf Seite 154 sendet der Adapter dem Auktionator nach Erhalt einer `newBid` Nachricht auf den Empfang einer `update` Nachricht, die vom Auktionator gesendet wird, eine entsprechende `updateAck` Antwortnachricht – auch falls der Bieter noch überhaupt keine `reqToBid` Anfrage gestellt hat. Somit werdem dem Auktionator die einer Versteigerung zu Grunde liegenden Interaktionen auch dann gestattet, wenn der Bieter (beispielsweise aufgrund technischer Probleme) nicht in der Lage ist, eine `reqToBid` Nachricht zu schicken und daher gar nicht an der Versteigerung teilnimmt. Auf diese Weise könnte die Kommunikation einseitig auf Seiten des Auktionators zu einem Endzustand gelangen – ohne dass der Bieter jemals begonnen hätte zu bieten.

Dieses Beispiel zeigt, dass der Entwickler eines Adapters sehr genau darauf achten sollte, ein möglichst »passives« Verhalten zu implementieren.[26] Nachrichten, die vom Adapter abgefangen werden, sollten möglichst an den eigentlichen Adressaten der Nachricht auf sinnvolle Weise weitergeleitet werden (notfalls sollte der Adapter mit dem Weiterleiten warten, falls die Weiterleitung nicht sofort möglich ist). Auf diese Weise lässt sich besser zusichern, dass die Kommunikation möglichst symmetrisch im Dreieck Adapter–Komponenten abläuft (vgl. Abbildung 3.6) – und somit ein auf Seiten der beiden Komponenten funktional erfolgreiches Ziel erreicht wird.

[26] Auch die explizite Etablierung eines *transaktionalen* Verhaltens kann hier nützlich sein.

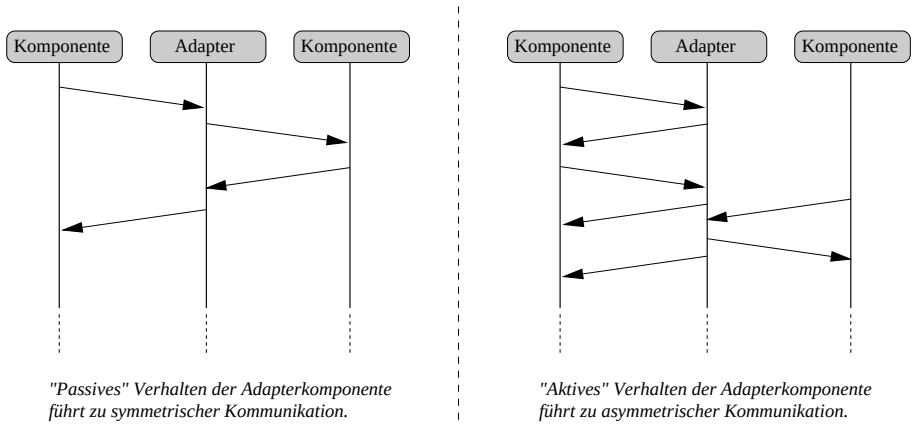


Abbildung 3.6 *Symmetrische und asymmetrische Kommunikation zweier Komponenten mit einer Adapterkomponente*

3.8 Prüfungsalgorithmus der Adapterkorrektheit

Ein Adapter, der gemäß den Regeln in Abschnitt 3.7.3 erstellt wird, lässt sich bezüglich seiner Kompatibilität zu den beiden kollaborierenden Komponenten prüfen.

Der Algorithmus dieser Kompatibilitätsprüfung ist dem Algorithmus der Kompatibilitätsprüfung in Abschnitt 3.6 sehr ähnlich. Der grundlegende Ablauf ist mit dem in Abbildung 3.3 illustrierten Ablauf identisch. Geprüft wird nun aber die Kompatibilität aller drei beteiligten Komponenten (des Adapters und je zwei der Komponenten, die kommunizieren möchten) zueinander.

3.8.1 Rückgabeinformationen

Das Prüfungsergebnis enthält Informationen, wie sie schon bei der Kompatibilitätsprüfung zweier Komponenten (vgl. Abschnitt 3.6.1) geliefert wurden. Auch bei diesem Prüfungsergebnis wird als Anzeichen einer Kompatibilität insbesondere auf das Nichtaufteten von unspezifiziertem Empfang und Verklemmung Wert gelegt. Bei *schwacher* Kompatibilität des Adapters (s. Abschnitt 3.8.2) wäre das Auftreten von unspezifiziertem Empfang teilweise erlaubt.

Neben diesen Kriterien, die die Protokollkompatibilität des Adapters beschreiben, muss vom Adapter gefordert werden, dass er zu keinem Zeitpunkt versucht, Parameter auszulesen und weiterzugeben, die zu diesem Zeitpunkt nicht in den Speicherzellen

vorliegen. Wenn diese Bedingung erfüllt ist, wird der Adapter als *speicherkonsistent* bezeichnet.

Ein Adapter, der speicherkonsistent und zu den beiden kollaborierenden Komponenten kompatibel ist, wird auch als *korrekt* (nur in Bezug auf die Interaktion mit genau diesen Komponenten) bezeichnet.

3.8.2 Unterscheidung schwacher und starker Kompatibilität

Gemäß Abschnitt 3.7.6 steuert ein Adapter die Kommunikation. Somit ist es ihm erlaubt Nachrichten zurückzuweisen, die ihn von einer der beiden Komponenten erreichen. Eine solche Situation stellt aus Sicht der Komponente auf den Adapter einen unspezifizierten Empfang dar.

In diesem Zusammenhang soll daher zwischen *schwacher* und *starker* Kompatibilität (engl. *weak* bzw. *strong satisfaction*) des Adapters zu den kollaborierenden Komponenten unterschieden werden. Starke Kompatibilität ist gegeben, wenn zu keinem Zeitpunkt der Kommunikation ein unspezifizierter Empfang auftritt. Schwache Kompatibilität erlaubt das Auftreten eines unspezifizierten Empfangs, wenn der Sender der Nachricht eine der beteiligten Komponenten, nicht aber der Adapter ist (Empfänger dieser Nachricht ist gemäß Abschnitt 3.7.6 der Adapter).

3.8.3 Ablauf des Prüfungsalgorithmus

Der Algorithmus entspricht in seinem Ablauf größtenteils dem in Abschnitt 3.6.2 beschriebenen Kompatibilitätsprüfungsalgorithmus für zwei Komponenten. Unterschiede bestehen in der Zusammensetzung der Tupel, aus denen die einzelnen Mengen bestehen und in der zusätzlichen Prüfung der Speicherkonsistenz des Adapters.

Betrachtet werden Zustandstupel $(\mathtt{s}, \mathtt{t}, \mathtt{a})$, wobei $\mathtt{s}$ und $\mathtt{t}$ die Zustände der beteiligten Komponentenprotokolle P_A und P_B und $\mathtt{a}$ der Zustand des Adapterprotokolls P_{Adap} zu einem betrachteten Zeitpunkt sind. Von einem Zustandstupel $(\mathtt{s}, \mathtt{t}, \mathtt{a})$ lässt sich auf folgende Arten ein Nachfolgezustandstupel herleiten.

Es existiert eine Nachricht $\mathtt{m}$, sodass einer der folgenden Fälle gilt:

- $\mathtt{a} : -\mathtt{m} \mapsto \mathtt{a}' \in \mathtt{P}_{\mathtt{Adap}}$ und $\mathtt{s} : +\mathtt{m} \mapsto \mathtt{s}' \in \mathtt{P}_{\mathtt{A}}$. Das Nachfolgezustandstupel ist somit $(\mathtt{s}', \mathtt{t}, \mathtt{a}')$

- $\mathtt{a} : -\mathtt{m} \mapsto \mathtt{a}' \in \mathtt{P}_{\mathtt{Adap}}$ und $\mathtt{t} : +\mathtt{m} \mapsto \mathtt{t}' \in \mathtt{P}_{\mathtt{B}}$. Das Nachfolgezustandstupel ist somit $(\mathtt{s}, \mathtt{t}', \mathtt{a}')$

- $\mathtt{s} : -\mathtt{m} \mapsto \mathtt{s}' \in \mathtt{P}_{\mathtt{A}}$ und $\mathtt{a} : +\mathtt{m} \mapsto \mathtt{s}' \in \mathtt{P}_{\mathtt{Adap}}$. Das Nachfolgezustandstupel ist somit $(\mathtt{s}', \mathtt{t}, \mathtt{a}')$

- $\mathtt{t} : -\mathtt{m} \mapsto \mathtt{t}' \in \mathtt{P}_{\mathtt{B}}$ und $\mathtt{a} : +\mathtt{m} \mapsto \mathtt{a}' \in \mathtt{P}_{\mathtt{Adap}}$. Das Nachfolgezustandstupel ist somit $(\mathtt{s}, \mathtt{t}', \mathtt{a}')$

Bei einem Zustandsübergang der Form $\mathtt{a} : -\mathtt{m} \mapsto \mathtt{a}'$ wird innerhalb des Algorithmus die Speicherkonsistenz des Adapters geprüft. Der Übergang ist ungültig (wird also nicht berücksichtigt), falls für diesen Übergang benötigte Parameter nicht ausgelesen werden können.

Bei Prüfung auf Existenz eines Nachfolgezustandstupels werden analog dem Kompatibilitätsprüfungsalgorithmus zweier Komponenten immer zunächst die Zustandsübergänge der beteiligten Komponenten (inklusive Adapter) auf gültige Übergänge mittels zu sendender Nachrichten geprüft. Erst bei Existenz eines solchen Übergangs wird die adressierte Komponente auf einen Zustandsübergang mittels der entsprechenden zu empfangenden Nachricht geprüft. Der zeitliche Ablauf der einzelnen Prüfung entspricht somit in der Reihenfolge den oben aufgeführten einzelnen Möglichkeiten.

Dieser Korrektheitsprüfungsalgorithmus kann im Ergebnis sowohl nicht-deterministische als auch deterministische Adapter (vgl. Abschnitt 3.7.4) als korrekt ansehen. Dies liegt darin begründet, dass der Algorithmus auch bei einem bereits gefundenen Nachfolgezustandstupel, das durch das Senden einer Nachricht seitens des Adapters erreicht wurde, nach weiteren Nachfolgezustandstupeln sucht.

Ein derartiges Vorgehen ist notwendig, denn falls die Suche nach Nachfolgezustandstupeln abgebrochen würde sobald ein mittels Sendevorgang des Adapters erreichtes Nachfolgezustandstupel gefunden wurde, bestünde die Gefahr, dass weitere Kommunikationsverläufe übersehen würden. Dies wäre fatal, wenn sich im weiteren Verlauf des Algorithmus herausstellt, dass der Kommunikationsablauf auf Basis des zuerst entdeckten Nachfolgezustandstupels beispielsweise in eine Verklemmung führt. Der Adapter würde somit als inkorrekt klassifiziert werden, obwohl andere Kommunikationsverläufe, deren Untersuchung unterblieben ist, eventuell in ein Zustandstupel mit Endzuständen hätte führen können, was ja ein wichtiges Kriterium der Kompatibilität der drei beteiligten Komponenten darstellt.

3.9 Semiautomatische Generierung von Adaptern

Die Erstellung der Regeln eines Adapters in der in Abschnitt 3.7.3 beschriebenen Form erscheint bei näherer Betrachtung kompliziert und daher fehleranfällig. Der jeweilige Entwickler muss sehr genau auf die Speicherkonsistenz seiner Regeln achten. Die Regeln wiederum stellen eine auf einer sehr niedrigen, technisch orientierten Ebene formulierte Spezifikationssprache für einen Adapter dar. Jedes mögliche Verhalten zu jedem Zeitpunkt muss detailliert in diesen Regeln formuliert sein und jeder mögliche Zustand, in den der Adapter geraten kann, berücksichtigt werden.

Insbesondere bei zunehmender Komplexität der kollaborierenden Komponenten wird die Anzahl der benötigten Regeln daher stark ansteigen. Zudem werden im Einzelfall Regeln redundant formuliert, da sie gleiche Kommunikationsverläufe mehrmals beschreiben müssen, falls sich die gültigen Speicherzellen von Fall zu Fall unterscheiden (und die neu hinzugekommenden oder weggefallenen gültigen Speicherzellen für diese Kommunikationsverläufe ohne Bedeutung sind).

Zur Vereinfachung der Adapterentwicklung wird im Folgenden daher eine Meta-Beschreibungssprache präsentiert, mittels derer sich die einzelnen Beziehungen zwischen Parametern und zwischen Nachrichten auf einer höheren Ebene konkret beschreiben lassen. Ausgehend von einer derart spezifizierten Adapterschnittstelle, lässt sich mittels des in Abschnitt 3.9.3 vorgestellten Algorithmus (die zugehörige Implementierung findet sich in Abschnitt 6.1.7) automatisch eine Liste von Regeln, die die in Abschnitt 3.7.3 erläuterte Form besitzen, erzeugen – falls dies möglich ist. Andernfalls wird der Algorithmus feststellen, dass die Adapterspezifikation sich nicht in einen Adapter, der mit den zu koppelnden Komponenten kompatibel ist, umsetzen lässt. Auf diese Weise steht ein *konstruktives* Verfahren bereit.

3.9.1 Eigenschaften von Adaptern

Die in Abschnitt 3.9.2 beschriebene Meta-Adapterspezifikation basiert darauf, dass ein Adapter, der seinerseits auf Regeln in der in Abschnitt 3.7.3 beschriebenen Form basiert, drei funktional voneinander unabhängige Eigenschaften besitzt, die sein Verhalten festlegen. Diese Eigenschaften lassen sich auch als Informationen über das Verhalten des Adapters bezeichnen. Im Einzelnen handelt es sich um folgende Informationen:

- Aus welchem (geladenen) Parameter wird welcher Parameter ausgelesen? Notiert wird `val` ▷ `parm`, wenn mindestens eine Adapterregel existiert, in der der Parameter `parm` einer zu sendenden Nachricht aus dem Wert eines bereits geladenden Parameters `val` ausgelesen wird.
- Wird ein empfangener Parameter einmal oder mehrfach benutzt, um zu sendende Parameter zu erzeugen? Wird der Wert eines Parameters `val` nur einmal ausgelesen, wird er der Terminologie nach [Yellin & Strom, 1995] folgend `val` als *oneshot*-Parameter bezeichnet – andernfalls als *multishot*-Parameter.
- Welche Kausalbeziehungen bestehen zwischen empfangenen und zu sendenden Nachrichten? Diese Information beinhaltet alle konkret möglichen Kommunikationsverläufe aus Sicht des Adapters.

Die formalen Regeln einer Adapterkomponente enthalten keine weiteren für den Ablauf der Kommunikation und das interne Speicherverhalten des Adapters benötigten Informationen.

3.9.2 Syntax der Meta-Adapterspezifikation

Die Meta-Adapterspezifikation gliedert sich gemäß den oben beschriebenen Eigenschaften von Adaptern in drei Abschnitte:

- `parm_mapping_rules` legen die Übergänge der Form `val` ▷ `parm` fest,
- `parm_using_rules` legen für einen bei einem Übergang zum Auslesen verwendeten Parameter die Eigenschaft `oneshot` fest,
- `causality_rules` legen die Kausalbeziehungen zwischen eingehenden und zu sendenden Nachrichten fest.

Die `causality_rules` bestehen aus regulären Ausdrücken. Ein derartiger regulärer Ausdruck wird hier als *pattern* bezeichnet, dessen Alphabet die Menge der Nachrichten ist, die vom Adapter empfangen oder gesendet werden können. Die Grammatik, mittels derer ein *pattern* konstruiert wird, ist die Folgende:

```
<pattern>  ::= <direction><msg_name>
             | <pattern> + <pattern>
             | <pattern> . <pattern>
             | <pattern> *
             | ( <pattern> )
```

Dabei kann `direction` die Werte »+« oder »–« annehmen und `msg_name` bezeichnet eine für den Adapter zu sendende oder empfangbare Nachricht. Wenn ein solcher Ausdruck die Form $\texttt{pattern}_1 + \texttt{pattern}_2$ annimmt, bedeutet dies, dass der Adapter die Nachrichtenabfolge senden und / oder empfangen muss, die durch $\texttt{pattern}_1$ *oder* $\texttt{pattern}_2$ erfüllt wird. Wenn der Ausdruck die Form $\texttt{pattern}_1.\texttt{pattern}_2$ hat, dann muss der Adapter zuerst die durch $\texttt{pattern}_1$ spezifizierte Nachrichtenabfolge senden und / oder empfangen und dann die durch `pattern2` spezifizierte Nachrichtenabfolge senden und / oder empfangen. Falls der Ausdruck von der Form $\texttt{pattern}*$ ist, so bedeutet dies, dass der Adapter die durch `pattern` spezifizierte Nachrichtenabfolge gar nicht oder mehrmals senden und / oder empfangen muss.

Zusätzlich ist die Notation `!(<pattern>)` erlaubt, welche den Adapter dazu zwingt zur Erfüllung des Ausdrucks andere als die in `pattern` spezifizierten Nachrichten zu senden oder empfangen.

Theoretisch lassen sich nun beliebig komplexe reguläre Ausdrücke als ein *pattern* formulieren, die ein Adapter erfüllen muss. In der Praxis werden hingegen meist Ausdrücke in der Art »wenn der Adapter eine Nachricht $\texttt{m}_1$ empfängt, muss er eine Nachricht $\texttt{m}_2$ senden« [27] (dies beschreibt eine 1:1 Beziehung zwischen zwei Nachrichten) oder Erweiterungen dieses Ausdrucks formuliert werden.

Da jeder reguläre Ausdruck auf einen endlichen Automaten abgebildet werden kann, bietet es sich an, eine Notation zu verwenden, wie sie in Abschnitt 3.3.1 für die Protokollvorschriften verwendet wird. Folgendes Beispiel beschreibt ein Protokoll für den soeben erwähnten Ausdruck:

```
A:  +m1  -> B; B:  -m2  -> A;
```

Hier nun die komplette Grammatik der Meta-Adapterspezifikation:

```
<MappingRule> ::= <parm_mapping_rule>
                | <parm_usage_rule>
                | <causality_rule>;

<parm_mapping_rule> ::=
  [<Function>]<parm_or_const>+ -> <parm>;

<parm_usage_rule> ::= one-shot <parm>;
```

[27] Das korrekte *pattern* zu diesem Ausdruck lautet $((!(\texttt{m}_1 + \texttt{m}_2)) * .\texttt{m}_1.(!(\texttt{m}_1 + \texttt{m}_2)) * .\texttt{m}_2) * .(!(\texttt{m}_1 + \texttt{m}_2))*$.

```
<causality_rule> ::= <pattern>;

<pattern> ::= <direction><msg>
            | <pattern> + <pattern>
            | <pattern> . <pattern>
            | <pattern> *
            | ( <pattern> );

<parm_or_const> ::= <constant>
                  | <parm>;

<parm> ::= <role_name>::<msg_name>.<parm_name>;

<msg> ::= <msg_name>::<role_name>;

<direction> ::= "+" | "-";
```

In einer `parm_mapping_rule` kann somit optional eine Funktion angegeben werden, die auf Grundlage des Wertes von `val` einen (anderen) Wert errechnet. Dies ist beispielsweise nützlich, wenn die über den Adapter gekoppelten Komponenten verschiedene Einheiten für ihre Parameter als Grundlage erwarten.

3.9.3 Algorithmus zur automatischen Generierung eines Adapterprotokolls

Ausgehend von einer Meta-Adapterspezifikation in der soeben beschriebenen Form lässt sich ein Adapter in der in Abschnitt 3.7.3 präsentierten Form erzeugen. Betrachtet werden soll zunächst als Beispiel für eine Meta-Adapterspezifikation die zu dem in Beispiel 3.6 auf Seite 154 dargestellten Adapter passende Meta-Adapterspezifikation:

```
//Parameter mapping rules
bidder::newBid.bidId -> auctioneer::bid.bidderId;
bidder::newBid.myBid -> auctioneer::bid.amount;
bidder::reqToBid.bidItem ->
  auctioneer::bid.itemBiddingOn;
auctioneer::newItem.itemDescr ->
  bidder::auctionBegin.auctionInfo;
auctioneer::newItem.bidderId -> bidder::canBid.bidId;
auctioneer::update.highBid -> bidder::newHighBid.price;

//One-shot rules
one-shot(bidder::newBid.bidId);
one-shot(bidder::newBid.myBid);
```

```
one-shot(auctioneer::newItem.itemDescr);
one-shot(auctioneer::newItem.bidderId);
one-shot(auctioneer::update.highBid);

//Causality rules
A: +newItem::auctioneer   -> B,
B: -auctionBegin::bidder  -> A;

A: +rejectBid::auctioneer -> B,
B: -bidNotOk::bidder      -> A;

A: +acceptBid::Auctioneer -> B,
B: -bidOk::Bidder         -> A;

A: +itemSold::auctioneer  -> B,
B: -gavel::Bidder         -> A;

A: +update::Auctioneer    -> B,
B: +reqToBid::Bidder      -> C,
C: -newHighBid::Bidder    -> D;
D: +itemSold::Auctioneer  -> A;

A: +highBidAck::Bidder    -> B,
B: -updateAck::Auctioneer -> A;

A: +reqToBid::Bidder      -> C,
A: +update::Auctioneer    -> B,
B: -updateAck::Auctioneer -> A,
C: +update::Auctioneer    -> D,
C: +itemSold::Auctioneer  -> A;
D: -newHighBid::Bidder    -> C;

A: +reqToBid::Bidder      -> B,
B: -canBid::Bidder        -> A;

A: +newBid::Bidder        -> B,
B: -bid::Auctioneer       -> A;
```

Beispiel 3.7 *Eine Meta-Adapterspezifikation*

Die `causality_rules` wurden in diesem Beispiel zur besseren Lesbarkeit in Form einzelner Protokollspezifikationen angegeben.

Aus einer solchen Auflistung, die auf der in Abschnitt 3.9.2 vorgestellten Syntax basiert, lässt sich nun unter Hinzunahme der Protokolle P_A und P_B der an der Gesamtkomposition beteiligten Komponenten (Auktionator und Bieter; vgl. Beispiele 3.1 und 3.5) auf folgende Weise ein Adapter generieren.

3.9.3.1 Voraussetzungen

In einer Meta-Adapterspezifikation existiert eine bestimmte Anzahl n an causality_rules. Da eine causality_rule einen endlichen Automaten beschreibt, besitzt ein auf Basis des regulären Ausdrucks erzeugter Automat zu einem bestimmten Zeitpunkt einen internen Zustand. Somit lässt sich zu jedem beliebigen Zeitpunkt eine Liste $(p_{r1}, ..., p_{rn})$ erzeugen, die sämtliche internen Zustände der den n causality_rules entsprechenden Automaten repräsentiert.

Im hier vorgestellten Algorithmus werden nun erneut (vgl. Abschnitte 3.6.2 und 3.8.3) einzelne Zustandstupel betrachtet, die jeweils den internen Zustand des Adapters zu einem bestimmten Zeitpunkt darstellen.

Es handelt sich hierbei um Zustandstupel der Form $(s_1, s_2, memCells, p_{r1}...p_{rn})$, wobei s_1 und s_2 die internen Zustände der Protokolle der beteiligten Komponenten sind, memCells eine Liste der gültigen Speicherzellen und p_{rx} (mit $1 \leq x \leq n$, wobei n=Anzahl der causality_rules) der Zustand des zur x. causality_rule gehörenden Protokolls ist.

Ein gültiger Übergang von einem Zustand $u = (s_1, s_2, memCells, p_{r1}, ..., p_{rn})$ zu einem Nachfolgezustand $u' = (s'_1, s'_2, memCells', p'_{r1}, ..., p'_{rn})$ mittels einer Nachricht m (formal: $u : \pm m \mapsto u'$) existiert genau dann, wenn folgende fünf Bedingungen erfüllt sind:

- $s_1 : -m \mapsto s'_1 \in P_1$ und $s_2 = s'_2$ *oder* $s_2 : -m \mapsto s'_2 \in P_2$ und $s_1 = s'_1$ (P_1 und P_2 bezeichnen die zu den beiden beteiligten Komponenten gehörenden Protokolle)
- Wenn m eine zu sendende Nachricht des Adapters ist, dann enthält memCells alle für die Erzeugung der Parameter von m notwendigen Parameterwerte.
- Wenn m eine zu sendende Nachricht des Adapters ist, dann enthält memCells′ die gleichen Werte wie memCells, außer das die zu m gehörenden Parameter, die als oneshot in den param_using_rules deklariert sind, fehlen.
- Wenn m eine vom Adapter zu empfangene Nachricht ist, dann enthält memCells′ alle Werte aus memCells und zusätzlich sämtliche zu m gehörenden Parameter.
- Der der x. causality_rule $(1 \leq x \leq n)$ entsprechende Automat führt einen Zustandsübergang $p_{rx} : \pm m \mapsto p'_{rx}$ durch, falls dieses im Protokoll der x. causality_rule vorgesehen ist.

Die erste Bedingung stellt sicher, dass ein Übergang des Adapterzustands nur möglicht ist, wenn kein unspezifizierter Empfang auf Seiten der beiden Komponentenprotokolle vorhanden ist. Die zweite Bedingung garantiert, dass eine Nachricht vom Adapter nur gesendet werden kann, wenn in `memCells` alle benötigten Werte zur Erzeugung der zu `m` gehörenden Parameter vorhanden sind. Die dritte und vierte Bedingung sorgen für eine korrekte Spezifikation des Zustandes von `memCells` nach einem Übergang. Die letzte Bedingung dient dazu, dass der Adapter seine intern auf Basis der `causality_rulse` implementierten Automaten gemäß der zugehörigen `causality_rules` weiterschaltet.

3.9.3.2 Ablauf des Algorithmus

Der Algorithmus geht bei der Konstruktion des Adapters in zwei Phasen vor: Zunächst wird ein Adapter `A0` konstruiert. Ein solcher Adapter kann Verklemmungen enthalten, die in der zweiten Phase sukzessiv entfernt werden, sodass am Ende ein korrekter Adapter verbleibt. Falls der Algorithmus letztlich zu einem »leeren« Adapter führt, also einem Adapter, der keinerlei Zustandsübergänge enthält, kann mit der dafür verwendeten Meta-Adapterspezifikation und den Protokollen der beteiligten Komponenten kein korrekter Adapter erzeugt werden. Insofern dient dieser Algorithmus auch der Prüfung der Korrektheit einer Meta-Adapterspezifikation in Bezug auf die beteiligten Komponentenprotokolle.

Die erste Phase verläuft nach dem gleichen Prinzip wie die Adapterkorrektheitsprüfung aus Abschnitt 3.8.3 (vgl. a. Abbildung 3.3 auf Seite 146) Es wird zunächst das Anfangstupel $(\mathtt{init}_{\mathtt{P1}}, \mathtt{init}_{\mathtt{P2}}, \emptyset, \mathtt{p}_{\mathtt{rinit1}}, \ldots, \mathtt{p}_{\mathtt{rinitn}})$ auf mögliche Zustandsübergänge hin untersucht, wobei $\mathtt{init}_{\mathtt{P1}}$ und $\mathtt{init}_{\mathtt{P2}}$ die Anfangszustände der Protokolle der interagierenden Komponenten sind und $\mathtt{p}_{\mathtt{rinitx}}$ (mit $1 \leq \mathtt{x} \leq \mathtt{n}$, wobei $\mathtt{n} = \mathtt{Anzahl}$ der `causality_rules`) der Anfangszustand des Protokolls der `x. causality_rule` in der Meta-Adapterspezifikation ist. Für Zustandsubergänge müssen die in Abschnitt 3.9.3.1 erörterten fünf Bedingungen erfüllt sein.

Neu gefundene Übergänge werden gespeichert, ebenso wie die damit neu erreichbaren Zustandstupel, die einer internen Menge `EQUIV`, die zunächst nur das Anfangstupel enthielt, hinzugefügt werden. Eine weitere Menge dient dem Speichern der neu erreichbaren Zustandstupel, die sich bisher noch nicht in `EQUIV` befanden.

Bei der Speicherung eines Zustandsüberganges werden nicht nur Ausgangs- und Zielzustandstupel zusammen mit der den Übergang komplettierenden Nachricht, sondern auch alle Informationen über das Speichern, Synthetisieren und Löschen von Parametern gespeichert.

Im nächsten Durchgang, der sich wiederholt bis kein Zustandsübergang mehr möglich ist, der zu einem Zustandstupel führt, das sich nicht in `EQUIV` befindet, werden die bei dem letzten Durchgang neu gefundenen Zustandstupel auf gültige Übergänge hin untersucht. Phase 1 endet, wenn im letzten Durchgang keine Zustandstupel erreicht werden, die nicht bereits in `EQUIV` vorhanden sind. Der Adapter `A0` ist nun komplett generiert.

In Phase 2 werden sämtliche gefundenen Zustandsübergänge daraufhin untersucht, ob das jeweilige Zielzustandstupel (das Zustandstupel, das mit diesem Übergang erreicht wird) wenigstens einmal in sämtlichen Übergängen des Adapters auch als Ausgangszustandstupel (ein Zustandstupel, das den Beginn eines Übergangs darstellt) verwendet wird. Ist dies nicht der Fall und handelt es sich beim betrachteten Zielzustandstupel auch nicht um ein Endzustandstupel (ein Zustandstupel, das von den Protokollen der beteiligten Komponenten jeweils einen Endzustand enthält), so stellt dieser Zustandsübergang eine Verklemmung dar und wird entfernt.

Somit entsteht eine gekürzte Liste von Zustandsübergängen, die nun wiederum auf Verklemmungen untersucht und gegebenenfalls weiter verkürzt wird. Phase 2 ist beendet, wenn bei der Untersuchung aller noch vorhandenen Zustandsübergänge keine Verklemmung entdeckt und entfernt wird. Die nun vorhandenen Zustandsübergänge bilden die Regeln für das Protokoll eines korrekten Adapters in der in Abschnitt 3.7.3 beschriebenen Form.

3.9.4 Ergänzung

Auffällig bei der Konzeption der Adapterprotokollspezifikation aus Abschnitt 3.7.3 und dem zugehörigen Beispiel 3.6 auf Seite 154 ist, dass es zwar konzeptionell vorgesehen ist, bei einer den Adapter *erreichenden* Nachricht das Löschen einzelner Speicherzellen (`memCells`) im Adapterprotokoll als `invalidate_actions` spezifizieren zu können, dies aber im angeführten Beispiel nicht angewendet wird. Daher ist im Rahmen der vorliegenden Arbeit noch eine Ergänzung zur Syntax der oben vorgestellten Meta-Adapterspezifikation konzipiert worden.

Das Nichtanwenden obiger Möglichkeit führt bei Berücksichtigung der Nachrichten `gavel` und `itemSold` (die in der Dar-

stellung des Beispiel gerade deshalb ausgelassen wurden) zu einer unnötig hohen Anzahl von Zustandsübergängen, da nach Erhalt einer `gavel` bzw. `itemSold`-Nachricht die beteiligten Komponenten zwar wieder in einen »niedrigen« Zustand (ein Zustand, der nahe beim Anfangszustand liegt) zurückkehren – der Adapter jedoch nicht, da zu seinem Zustandstupel auch die gültigen Speicherzellen `memCells` gehören. Er gerät an dieser Stelle somit in einen Zustand, in dem er »nutzlose« Informationen speichert, die in den folgenden Zustandsübergängen vorraussichtlich überschrieben werden, denn die Nachrichten `gavel` und `itemSold` dienen dem Beenden der Auktion eines Objektes und ermöglichen die Auktion eines neuen Objektes.

Es wäre somit sinnvoler, wenn das Löschen von Speicherzellen (in diesem Falle aller Speicherzellen) auch beim Empfang einer Nachricht seitens des Adapters spezifiziert wird. Eine deutliche Reduktion der Anzahl der Zustandsübergänge im Protokoll des Adapters wäre die Folge mit einer entsprechenden Verbesserung der Praktikabiltität.

Während diese Möglichkeit bei einer direkten Adapterprotokollspezifikation potenziell gegeben ist, ergibt sich für die Meta-Adapterspezifikationssyntax aus Abschnitt 3.9.2 eine Lücke. Hier ist das Löschen einzelner Speicherzellen lediglich vorgesehen, wenn ein als `oneshot` deklarierter Parameterübergang (definiert durch `parm_using_rule` und `parm_mapping_rule`) in einem Zustandsübergang stattfindet.

Das explizite Löschen beliebiger Speicherzellen außerhalb eines derartigen Parameterübergangs ist in diesem Modell nicht vorgesehen. Somit können weder Speicherzellen gelöscht werden, die bei einem speziellen Parameterübergang nicht selbst beteiligt sind, noch ist das Löschen von Speicherzellen beim *Empfang* einer Nachricht (beispielsweise `itemSold`) vorgesehen.

Korrekte Adapter werden mittels des Algorithmus in Abschnitt 3.9.3 zwar erzeugt, jedoch benötigen sie möglicherweise »unnötig« viele Zustandsübergänge. Eine Verbesserung könnte somit in der Meta-Adapterspezifikationssyntax ansetzen und das Löschen bestimmter Parameter bei Empfang (oder bei Versendung) bestimmter Nachrichten erlauben. Am sinnvollsten ist eine Ergänzung der `causality_rules` zu diesem Zweck.

```
... <causality_rule> ::= <pattern>;

<pattern> ::= <direction><msg><invalidate_actions>
            | <pattern> + <pattern>
            | <pattern> . <pattern>
            | <pattern> *
```

```
            | ( <pattern> );
...
```

Einem `pattern` wird somit die optionale Angabe von `invalidate_actions`, die wiederum eine Liste von zu löschenden Parameterwerten enthalten, zu jeder zu empfangenden oder zu sendenden Nachricht erlaubt. Der Algorithmus zur automatischen Generierung der Adapterregeln kann derartige Angaben bei Erzeugung eines Adapters `A0` berücksichtigen, dass heißt die entsprechende Eliminierung der Parameterwerte durchführen.

3.10 Attributbasierte Adaption

Die bisherige Betrachtung einer Adaption ging von einer Angleichung des Verhaltens einer Komponente aus. Eine Anpassung kann jedoch auch lediglich auf die Modifikation von Komponenteneigenschaften im Sinne von Attributen abzielen. Einfache Beispiele wären etwa die Angleichung der verwendeten Währung im Falle der angedeuteten Auktionskomponenten oder eine farbliche Anpassung gemeinsam zu verwendender GUI-Komponenten.

Insbesondere erlaubt eine attributbasierte Adaption jedoch nicht nur eine Anpassung von Komponenten *aneinander*, sondern auch die Einhaltung genereller Rahmenbedingungen und Richtlinien, etwa der Form »biete niemals mehr als 1000 Euro« oder »bette rote GUI-Komponenten niemals in blaue Fenster ein«. Dies erlaubt damit vor allem auch die Berücksichtigung spezifischen Fachwissens über einen Anwendungsbereich und kommt wiederum der fachnahen Sicht des Ansatzes dieser Arbeit entgegen. Die Anpassung auf Basis solcher Richtlinien (engl. *Policies*) erlaubt in diesem Sinne auch ein »*Product Alignment*«, d.h. zum Beispiel eine kundenspezifische Produktanpassung, bzw. eine konsistente Etablierung der »*cross-cut*« Eigenschaften einer Applikation (vgl. Abschnitt 5.5.1.1).

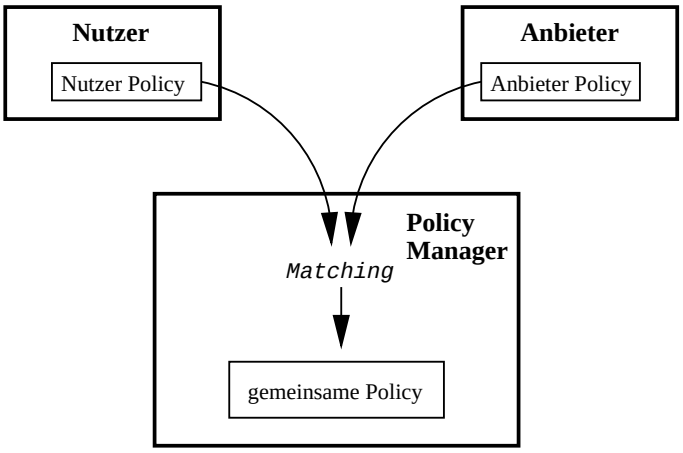
Der Vorteil des vorgestellten Ansatzes liegt dabei darin, dass jede Komponente zunächst beliebige (individuelle) Richtlininen festlegen kann, die für sie in Frage kommen. Welche daraufhin tatsächlich im Gesamtsystem (Kompositum) berücksichtigt werden müssen, hängt dann von der Unifikation mit anderen Komponentenrichtlinien ab und wird nicht von außen in das System induziert.

Mechanismen zur Verwaltung generischer Richtlinien [Tu et al., 1997] stellen eine relativ neue Form der Unterstützung interaktionsbasierter und insbesondere verteilter Systeme dar. Ausgangspunkt sind ganz allgemeine Anforderungen, die

von den Teilnehmern an die Art der Kooperation gestellt werden. Bei einer Anwendung dieses Konzeptes auf komponentenorientierte Systeme sind im Kontext dieser Arbeit zum einen erweiterte Anforderungen an die logischen Repräsentationen solcher Policies von Interesse, zum anderen deren Relation zu konkreten Komponentenmodellen.

Als Instrument zur Formulierung entsprechender Anforderungen kommen generische Richtlinien zum Einsatz, wobei die vorliegende Arbeit auf den umfangreichen Arbeiten von [Tu, 2000] aufsetzt. Entsprechend sollen hier nicht die eigentlichen Formalismen resümiert werden, sondern lediglich auf die entsprechenden Erweiterungen und Implikationen für eine Komponentensicht eingegangen werden. Policies werden durch logische Formeln in schlanker disjunktiver Normalform der Prädikatenlogik erster Ordnung beschrieben, deren wesentliche Terme *Properties* sind. Bei diesen Properties handelt es sich um konkrete Objekteigenschaften, die separat innerhalb eines *Propertymanagers* [Schmidt, 1997] verwaltet werden. Es bietet sich nun eine Erweiterung von Policies um temporallogische Konzepte gemäß der Darstellung im vorangegangenen Kapitel an. Dadurch können Policies auch *kausale* Aussagen über Properties berücksichtigen. Ferner sollten die Properties direkt mit den entsprechenden Attributen einer Komponente assoziiert werden.

Abbildung 3.7
Prinzip des Policymanagements



Ein als *Policy-Manager* [Göllnitz, 1997] bezeichneter Mechanismus verwaltet dann die Richtlinien und ist in der Lage, sie *automatisch* zu evaluieren. Ferner können durch das *Policy-Management* die Aussagen verschiedener Richtlinien in *Einklang* gebracht (durchgesetzt bzw. unifiziert) und in einer neu

generierten Policy zusammengefaßt und mit einer Komponente assoziiert werden. Da eine Komponente durch Registrierung entsprechender Eigenschaften beim Policy-Manager im Falle der Durchsetzung einer Richtlinie konkret durchzuführende *Aktionen* angeben kann (in Form von Aufrufen auf sie selbst), kann ein Komponentenentwickler entsprechend konsistente Manipulationen der Komponente vorsehen und dennoch eine benutzerseitige oder eben *automatische* Anpassung zulassen ohne dabei die Autonomie der Komponente zu beeinträchtigen.

Eine vollständige Integration des Policy-Managements in diesem Sinne wurde im Rahmen dieser Arbeit aufbauend auf [Tetau, 1999] vorgenommen – der ursprüngliche »Extended Policy Manager« wird daher nun als »Component Policy Manager« bezeichnet. Auf die in der originären Richtlinienverarbeitung vorgesehene *Schlichtung* im Falle nicht unifizierbarer Eigenschaften wird allerdings bewusst verzichtet, da entsprechende Entscheidungen ein Abweichen von ursprünglich durch den Komponentenentwickler in seiner Komponente verankerten Kriterien bedeuten und damit die geforderte Autonomie einer Komponente verletzen würde.[28] Das ursprüngliche Typsystem des Policy-Managments musste um das in dieser Arbeit zu Grunde gelegte Basistypsystem (entsprechend dem CORBA 3.0 Standard) erweitert werden.

Die Komposition von Komponenten führt zur Auswertung ihrer Richtlinien und der Erstellung einer *neuen* gemeinsamen Richtlinie für das entstandene Kompositum. Auf diese Weise ist zwar ein homogenes Modell im Sinne aggregierter Komponenten gegeben, allerdings muss das Policy-Management nun in der Lage sein, entsprechende »Zwischenzustände« während der Anwendungskompsition zu speichern. Auch dann verbleibt jedoch das Problem, das Komponenten nicht ohne weiteres wieder aus einem Aggregat *entfernt* werden können, soll eine erneute komplette Unifikation aller verbleibenden Komponenten vermieden werden. Im Gegensatz zum Einsatz in interaktiven Entwicklungsumgebungen (vgl. [Tetau, 1999]) stellt dies jedoch für den hier beabsichtigten Einsatz durch einen Generator kein Problem dar, da zum Zeitpunkt des entsprechenden Konstruktionsvorgangs die zu verwendenden Komponenten bereits selektiert wurden.

Dies ist auch der Grund, warum Policies nicht gleichberechtigt mit anderen Constraints (vgl. das nächste Kapitel) in den

[28] Treten entsprechende »Problemfälle« im Rahmen des automatischen Konstruktionsvorgangs auf, werden sie zur manuellen Entscheidung im Rahmen des Konfigurationsdialogs mit dem Anwendungskonstrukteur angeboten (vgl. a. Kapitel 5).

Typbegriff einer Komponente einfließen sollten, da die *Aktivierung* einer Richtlinie, also deren Durchsetzen, zum einen eine aktive Komponenteninstanz voraussetzt, zum anderen aber insbesondere nicht ohne weiteres rückgängig gemacht werden könnte, da sie möglicherweise zu internen Zustandsänderungen der Komponente geführt hat. Daher wäre auch die Realisierung eines Lösch-Konzepts für Aggregate etwa durch Speichern aller während der Komposition intermediär auftretenden Richtlinien nicht hinreichend. Ein Überblick über das Gesamtkonzept findet sich in [Griffel et al., 1999], sodass im folgenden noch ein kurzer Bezug zu anderen Ansätzen erfolgen soll, die den Einsatz einer Richtlinienverarbeitung im Rahmen einer komponentenbasierten Anwendungskonstruktion zusätzlich motiviert haben.

Policies können zum einen insbesondere in Analogie zu [Dingel et al., 1997] als Hilfsmittel für einen *Integrationstest* einer komponentenbasierten Anwendung dienen. Auch das »Design Rule Checking« von [Batory & Geraci, 1995] kann mit einer Policy-überwachten Komposition realisiert werden. Die von [Smaragdakis, 1999] postulierten »Proposition Properties« können in direkter Analogie zu dem hier vorgeschlagenen Einsatz von Policies betrachtet werden, wobei die hier verwendeten Richtlinien allerdings den Vorteil besitzen, auch nichthierarchische Prüfungen zu erlauben. [Batory, 1997] realisiert die von ihm geforderte geforderte »Intelligenz« einer Komponente über Attributgrammatiken, die für seine Quellcode-orientierte Sicht auch geeignet erscheinen. Eine entsprechende Implantierung von Policies in den Komponenten erlaubt allerdings eine deutlich anwendungsnähere Sicht, wie sie auch vom hier vorgestellten Ansatz präferiert wird. [Shaw, 1996] schlägt »Credentials« zur Verhaltensbeschreibung von Komponenten vor. Diese sollen zum Beispiel Aussagen über die Zuverlässigkeit der eigenen Informationen zulassen. Dies kann durch Einführung zusätzlicher entsprechender Attribute auch mit Hilfe der Richtlinienverarbeitung realisiert werden. Credentials geben allerdings nur einen *äußeren Rahmen* an Bedingungen vor (zum Beispiel wie gut wurde die Zusicherung getestet). Der hier vorgestellte Ansatz verknüpft diese Merkmale jedoch direkt mit der jeweiligen Komponente.

3.11 Bewertung

Die vorgestellen Konzepte zur Unterstützung einer operationalen, d.h. hier protokollbasierten Komponenteninteraktion und

deren dynamischer Adaption stellen einen nützlichen Beitrag zur Verbesserung und Prüfung der Kopplungsfähigkeit von Komponenten und damit einer Verbreiterung deren Einsatzmöglichkeiten dar. Hervorzuheben ist dabei – insbesondere im Gegensatz zu vergleichbaren Konzepten wie in [Okumura, 1986] – die Berücksichtigung der Parametrisierung der auszutauschenden Nachrichten.

Es handelt sich letztlich um eine praktische Umsetzung des Grundgedankens von [Nierstrasz, 1995b] zur Typanreicherung mittels operationalen Verhaltens, allerdings kritisiert [Allen, 1997] die Beschränkung solcher Ansätze auf (Objekt-) Methoden. Die hier am Beispiel der Methodenaufrufe von Kopplungspartnern dargestellten Mechanismen können jedoch durch die Interpretation als allgemeine »Nachrichten« auch im Rahmen eines allgemeineren Typmodells sinnvoll eingesetzt werden, wie das nächste Kapitel zeigen wird.

Die Adapter erlauben dem Ansatz dieser Arbeit dabei insbesondere die Abschwächung der Einschränkung von »provide / require«-Aussagen in den Schnittstellen üblicher Komponentenmodelle [Allen, 1997], indem sie explizite Konnektoren darstellen, die auch unabhängig von einzelnen Komponentenspezifikationen ein Interaktionsmuster und damit ein Konnektorverhalten im Architektursinne liefern. Dennoch sollte die Angebots- / Nutzungssemantik für Schnittstellenspezifikationen beibehalten werden, da nur auf diese Weise eine Abbildung allgemeiner Interaktionsmuster gegen eine *konkrete* Komponente möglich wird. Insbesondere erlaubt die Adaptergenerierung (bzw. deren Bereitstellung durch ein Typmanagement – siehe nächstes Kapitel) die Erzeugung neuer Konnektoren auf Basis der Interaktionsmöglichkeiten bestehender Komponenten ohne manuellen Eingriff während der Konstruktionsphase einer Applikation. Zudem vermeidet diese Lösung die in anderen Ansätzen auftretenden *asymmetrischen* Interaktionsmodelle (vgl. etwa [Magee et al., 1995]), die ein bestimmtes Interaktionsverhalten jeweils nur einem Initiator vorbehalten, indem hier jede beteiligte Komponente ein vollständiges Interaktionsmuster trägt.

Des Weiteren ist als positiv zu vermerken, dass die im Rahmen der erläuterten Konzepte auftretenden Protokolle, Adapter und Meta-Adapter eine einfach zu implementierende formale Überprüfung ihrer Korrektheit (bzw. Kompatibilität) zulassen.[29] Auch andere Autoren halten den Einsatz von Automa-

[29] vgl. etwa a. die modellbasierte Verifikation durch Automaten in [Gluch & Weinstock, 1998]

ten trotz der möglichen Gefahr von »Zustandexplosionen« für gerechtfertigt, da durch die Lokalisierung der Tests der vorgestellten Algorithmen entsprechend »kleine« Automaten entstehen [Allen, 1997]. Allerdings wird durch eine entsprechende Segmentierung der Komponentenschnittstellen und eine Hierarchiebildung dieser Gefahr bei der nachfolgenden Verwendung dieser Mechanismen zusätzlich vorgebeugt.

Da eine operationale (interaktionsorientierte) Sicht auf Komponenten angestrebt wird, ist die Auswertung von Vor- und Nach*zuständen* der entsprechenden Protokollautomaten ausreichend. Die Einführung expliziter Vor- und Nach*bedingungen* ist daher nicht zwangsläufig notwendig, inbesondere da diese in der Regel bereits Interna einer Komponente beträfen, die wenn überhaupt wiederum nur über Methodenzugriffe zugänglich sein sollten. Bei Bedarf (bzw. zur Simulation anderer Modelle) können jedoch die vorgestellten Policies zur Darstellung entsprechender Bedingungen über Attribute der Komponente eingesetzt werden. Die von [Allen, 1997] geforderte Verklemmungsfreiheit für Konnektoren wird ebenfalls vom vorgestellten Ansatz geleistet, sodass in Allens Sinne sowohl eine Stilkonsistenz als auch eine Konfigurationskonsistenz überprüfbar ist bzw. gewährleistet werden kann, letztere insbesondere auch durch das Einbeziehen der attributbasierten Anpassung.

Dabei bleibt jedoch zwangsläufig insbesondere für die Resultate einer Kompatibilitätsprüfung zweier Protokolle (mit oder ohne Adapter) Interpretationsspielraum. Ob zwei Komponenten, deren Prüfungsresultat das Erreichen von Endzuständen ermöglicht, *immer* (bei jedem denkbaren Kommunikationsverlauf) kompatibel zusammenarbeiten werden, lässt sich gegebenenfalls nicht feststellen. Ebenso könnten Komponenten, deren Prüfungsresultat auf unspezifizierten Empfang oder Verklemmung hinweist, trotzdem möglicherweise während der gesamten Laufphase einer Anwendung funktional korrekt zusammenarbeiten. Dies legt einen mehrstufigen Einsatz operationaler Beschreibungen nahe, sodass im Zweifelsfall eine negative Entscheidung auf einer höheren Ebene die (möglicherweise) nicht vollständige Entscheidbarkeit auf einer darunterliegenden Ebene gar nicht erst sichtbar werden lässt. Die Konzeption des Typmodells im nachfolgenden Kapitel wird dies berücksichtigen.

Adapter sind im beschriebenem Konzept im Sinne einer homogenen Einbettung in das grundlegende Komponentenmodell des Kapitel 2 spezialisierte Komponenten, die lediglich für die Zusammenarbeit mit den dazu bestimmten Komponenten geeignet sind. Eine vollautomatische Adaptererzeugung ist dabei un-

möglich, da ein Adapter ein »Wissen« über die *Bedeutung* einzelner Methodenaufrufe benötigt, das ihm nur der Komponentenentwickler zur Verfügung stellen kann. Durch den Einsatz einer Meta-Adapterspezifikation wird dieser notwendige manuelle Arbeitsschritt von automatisierbaren Arbeitsschritten getrennt. Dem Komponentenentwickler bietet sich so eine komfortablere Möglichkeit der Angabe der zur Adaption benötigten Funktionalität auf einem hohen Abstraktionsniveau, ohne dass er mit (technischen) Problemen der darunter liegenden Ebenen konfrontiert und belastet wird – insbesondere wenn die Handhabung dieser Ebene von einem entsprechenden Werkzeug unterstützt wird (vgl. a. Abschnitt 6.4.2).

Mittels der im Rahmen des Adaptionskonzeptes entwickelten Protokolle steht nun insbesondere eine operationale Form einer Verhaltensspezifikation zur Verfügung, deren Kompatibilitätsprüfungen im Rahmen eines Klassifikationskonzeptes für Komponenten ausgenutzt werden kann. Der Entwurf eines solchen Konzeptes als Grundlage einer späteren automatisierten Suche, Auswahl und eventuellen notwendigen Adaption von Komponenten im Rahmen einer generativen Softwarekonstruktion ist Gegenstand des folgenden Kapitels.

4 Klassifikation

Nachdem in den vorangegangen Kapiteln die Aspekte eines Komponentenmodells mit seinen wünschenswerten Eigenschaften sowie eine Möglichkeit zur Adaption entsprechender Komponenten dargestellt worden sind, sollen die zugehörigen Ergebnisse nun genutzt werden, um ein Klassifikationskonzept für Softwarebausteine zu konzipieren. Zur Veranschaulichung des dabei vorgestellten Typmodells finden sich in Anhang D konkrete Beispiele entsprechend typisierter Komponenten, die ein Nachvollziehen der einzelnen auftretenden Beziehungen und Elemente des Modells erleichtern sollen.

4.1 Vom Modell zur Laufzeitkomponente

Dem Wunsch nach einer Wiederverwendbarkeit von Software steht immer noch der Mangel an ausreichenden Abstraktionen in der Entwicklung einer Software entgegen. Zur Etablierung entsprechend notwendiger Abstraktionsebenen weist die Objektorientierung mit dem aufsetzenden Komponentenparadigma derzeit auf die Relevanz von *Meta-Modellen* [Griffel, 2000a][OMG, 1999b] hin. Wenn zudem der Komponentenbegriff selbst bereits als (möglicherweise ausreichende) Abstraktionsform diskutiert wird, stellt sich die Frage, wie und in welcher Form er von solchen *Metaebenen* profitieren kann bzw. in welchem Verhältnis er dazu steht.

Ein erster naheliegender Gedanke besteht hier in der vertieften Betrachtung einer *Typisierung* von Softwarekomponenten, um deren Klassifikation nach Struktur und Verhalten (vgl. Kapitel 2 und damit insgesamt ihren *Nutzungsmöglichkeiten* zu ermöglichen. Die zu der Komponente korrespondierende Typbeschreibung kann dann als ihr Meta-Modell interpretiert werden, das der jeweiligen Komponente eine *Selbstbeschreibung* erlaubt.

Die Ableitung von Beziehungen zwischen Komponenten auf Basis solcher Beschreibungen soll im weiteren als Typmanagement bezeichnet werden. Die dazu notwendige Funktionalität,

die sich primär auf die Untersuchung von *Konformitäts*- sowie *Kompatibilitätsbeziehungen* zwischen Komponententypen stützt, soll durch einen *Typmanager* bereitgestellt werden.

Die Beschreibung eines Komponententyps stützt sich dabei auf die aus der Komponentenmodellierung bzw. dem zugehörigen Domain Engineering hervorgegangenen Metainformationen. Da das Komponentenparadigma einerseits von der Auslagerung von (insbesondere nicht-funktionalen) Informationen aus der eigentlichen Komponente heraus[1] ausgeht, andererseits aber die direkte Assoziation solcher Zusatzinformationen mit einer Komponente fordert, ist das derzeit favorisierte Konzept der *Komponentendeskriptoren* entstanden (vgl. a. [Sun, 2000],[OMG, 1999]). Da ein solcher Deskriptor von den Entwicklungs- und Laufzeitumgebungen entsprechender Komponenteninfrastrukturen im Sinne einer deklarativen Programmierung ausgewertet wird, könnte er einen bruchlosen Übergang zwischen Modellierungs- und Laufzeit schaffen, wenn er ebenso als Träger der typrelevanten Metainformationen dient.

Die wesentliche Motivation dieser (Selbst-) Beschreibung von Komponenten besteht dabei in der Etablierung einer aussagekräftigen Spezifikation als *integralem Bestandteil* des Softwarebausteins, um überprüfbare sowie konsistenzsichernde Implementierungen bereits in der Entwicklungsphase einer Software zu gewährleisten. Entsprechende Ausprägungen von Spezifikationsmöglichkeiten hat Kapitel 2 erörtert.

Letztendlich besteht der Wunsch nach einem plug-and-play-Einsatz von Softwarebausteinen, wobei lediglich die Konfiguration der Komponente vor ihrem Einsatz an explizit dafür vorgesehenen Konfigurationspunkten vorgenommen werden muss. In diesem Kapitel steht die dafür als notwendig erachtete Konzeption und Formalisierung eines ausdrucksmächtigen Typmodells im Vordergrund.

4.1.1 Ansätze der Typisierung

Formalisierung durch Typisierung stellt eine Basis für die Darstellung von *Beziehungen* (*engl. relations*) zwischen Komponenten dar. Solche Beziehungen sind vor allem in dynamischen Umgebungen, in denen eine ständige Evolution und Fluktuation von Komponenten stattfindet, von entscheidender Wichtigkeit, da sie auch unter diesen erschwerten Bedingungen eine geregelte, typsichere Komponentennutzung erlauben. Beziehungen können in

[1] zwecks besserer Wiederverwendung des funktionalen Kerns

Abhängigkeit von den Intentionen bei ihrer Schöpfung und der Mächtigkeit ihrer Aussagen stark unterschiedlichen Charakters sein. Im wesentlichen sind dabei zwei Formen wichtig:

- *Durch Instanzen definierte Beziehungen*, die explizit bestimmt werden müssen (meist semantischer Natur).
- *Durch Definitionsregeln definierte Beziehungen*, die eine automatische Verifikation mittels formaler Regeln ermöglichen.

Eine erste relevante Beziehung ist die *Konformität* zwischen Komponententypen, da sie für das Auffinden geeigneter bzw. alternativer Komponenten in komplexen Systemumgebungen entscheidend ist. Wird etwa der Komponententyp A benötigt, kann ebenso ein möglicherweise erweiterter Baustein des Typs B genutzt werden, falls dieser konform zu Typ A ist, da sich die beiden Einheiten unter diesen Bedingungen aus Nutzersicht nicht im Verhalten unterscheiden. Der Typ A wird dann als *Supertyp* des *Subtyps* B bezeichnet. Die Subtypisierung erlaubt den Aufbau von *Typhierarchien* als Strukturen, welche die Konformitätsbeziehungen widerspiegeln. Die Konformitätsbeziehung gehört zu den durch formale Regeln definierbaren Beziehungen, was automatisierte Typvergleiche ermöglicht. Letzteres ist Voraussetzung für die generelle Praxistauglichkeit dieses Konstruktes, da in komplexen Systemen eine große Anzahl solcher Konformitätsvergleiche anfallen.

Neben Konformitätsbeziehungen ist aus komponentenorientierter Sicht die Verträglichkeit von Komponententypen entscheidend, was als *Typinteroperabilität* oder auch *-kompatibilität* bezeichnet werden könnte. Eine solche Typinteroperabilität drückt dann die Fähigkeit des Zusammenwirkens von Komponenten verschiedenen Typs bezüglich Struktur und Semantik aus.

Ein Problem besteht darin, dass herkömmliche Typsysteme oft nicht ausdrucksstark genug sind, um alle wichtigen Aspekte einer Komponenten*interaktion* zu erfassen. Es bieten sich verschiedene Lösungsstrategien an. Nierstrasz [Nierstrasz & Tsichritzis, 1995] schlägt etwa die zwei folgenden vor:

- *Schnittstellenerweiterung* – Die Schnittstellen von Komponenten können durch *Constraints* – also formale Bedingungen – erweitert werden[2], die zusätzliche Forderun-

[2] siehe z.B. Meyer [Meyer, 1988]: »programming by contract« in der Programmiersprache *Eifel*

gen und Erwartungen der beteiligten Kooperationspartner ausdrücken. Solche Constraints können zum Teil statisch durch Typsysteme und zum Teil dynamisch zur Laufzeit überprüft werden.

- *Ausdrucksstärkere Typsysteme* – Ausdrucksstarke Typsysteme existieren vor allem im Bereich funktionaler Sprachen. Wie bereits in Kapitel 2 erörtert sind die Möglichkeiten formaler Verifikation bei verteilten Mechanismen wie etwa einer Blackboard-Komposition weitaus geringer. Bei der Typisierung objektorientierter Systeme ergeben sich dabei kaum zu überwindende Probleme [Fisher & Nielsen, 1994]. Dies liegt vor allem in der temporalen Sensitivität von Objektzuständen, in der rekursiven Semantik und im Subtypisierungskonzept des Paradigmas begründet.[3] Diesen Schwierigkeiten wird meist mit existentieller Quantifikation begegnet, die aber dem Anwender eine extrem umständliche Handhabung aufbürdet und großen Berechnungsaufwand bedingt, der eine praktikable Umsetzbarkeit in konkreten Implementationen in der Regel verhindert.

4.1.1.1 Verhältnis zum Spezifikationsbegriff

Komponentenspezifikationen dienen gemäß Kapitel 2 in dieser Arbeit der Darstellung von Informationen eines Komponentenmodells in abstrakter Weise und bilden damit auch die Basis einer Typisierung.[4] Verschiedene Formen von Komponentenspezifikationen unterscheiden sich durch ihre Ausdrucksmächtigkeit sowie die daraus resultierenden Möglichkeiten automatisierter Vergleiche einzelner Beschreibungen. Grundsätzlich werden drei Gruppen möglicher Komponentenspezifikationen unterschieden:

Namensbasierte Spezifikation Bei namensbasierter Spezifikation werden Komponententypen nur durch ihre Namen beschrieben. Eine Erkennung und Überprüfung von Typbeziehungen ist hier nur explizit durch manuelle, administrative Festlegung möglich. Das Sicherstellen von Interoperabilität ist unter diesen Umständen problematisch, da ein globales Verständnis der Bezeich-

[3] Diese Erkenntnisse haben den Autor bewogen, ein *interaktionsbasiertes* Typkonzept zu propagieren, wie es in dieser Arbeit vorgestellt wird.

[4] Davon unberührt können selbstverständlich für einzelne Ausdruckselemente einer Spezifikation ebenfalls Typsysteme bestehen.

ner ohne weitreichende Standardisierungsbemühungen kaum zu erreichen ist. Namensbasierte Spezifikationen kommen auf Grund ihrer einfachen Realisierbarkeit zum Beispiel in den meisten verteilten Systemplattformen wie DCE, CORBA oder ANSA zum Einsatz. Ein Beispiel sind die seit der Einführung des Distributed Computing Environments (DCE) häufig anzutreffenden UUIDs (*universal unique identifier*) zur Kennzeichnung von Schnittstellen. Die Vergabe solcher weltweit eindeutigen *Nummern* verdrängt dabei zugleich die Frage nach eindeutigen *Bennenungen*.

Strukturelle Spezifikation Bei einer strukturellen Spezifikation enthalten die Beschreibungen lediglich Strukturaspekte der Komponenten in Form von herkömmlichen, zumeist signaturbasierten Schnittstellenbeschreibungen. Beziehungen zwischen Komponenten können so – zumindest auf dieser strukturellen Ebene – auch automatisiert verifiziert werden. Schwachstelle struktureller Spezifikationsformen ist das Fehlen eindeutiger semantischer Aspekte. So ist es möglich, dass strukturgleiche Komponenten doch ein voneinander abweichendes Verhalten zeigen, und eine durch die Struktur implizierte scheinbare Konformität bei Berücksichtigung der Semantik keineswegs vorhanden ist wie Kapitel 3 gezeigt hat.

Erweiterte Ansätze Typsysteme in Form von algebraischen Spezifikationen drücken lediglich statische Aspekte aus, führen aber bei der Untersuchung rein statischer Konfigurationen von Komponenten zu aussagekräftigen Ergebnissen. Im Falle von Komponentensystemen sind jedoch auch *dynamische Aspekte* von Bedeutung, die in dieser Form nicht formuliert werden können. So ist bei einer Komponente, die gewisse Methoden zur Verfügung stellt, nicht nur deren Ausprägung wichtig, sondern auch deren kausale Abhängigkeiten – also etwa die mögliche Reihenfolge der Anwendung – von Bedeutung (vgl. Kapitel 3. Solche Formen des dynamischen Verhaltens lassen sich durch *denotationale-*, *algebraische-*, *operationale-* und *axiomatische Semantiken* ausdrücken. Problematisch ist dabei die hohe Komplexität solcher Spezifikationsformen, die oft nicht einmal entscheidbar sind. Es sind in daher praktikable Kompromisse in Form von Zwischenlösungen anzustreben. Als eine solche wird der im weiteren beschriebene Ansatz verstanden, der sich entsprechend zwar Elemente aller drei Herangehensweisen bedient, aber auf eine pragamtische Handhabbar- und Umsetzbarkeit in die Entwicklungspraxis abzielt.

4.1.1.2 Gegenstände der Typisierung

Unter Typisierung wird im allgemeinen die Zusicherung einer invarianten Eigenschaft im Sinne eines Prädikats für ein bestimmtes Objekt, das dann eben *von* diesem Typ *ist*, verstanden. Es stellt sich dann die Frage, welche Eigenschaften einem Typsystem zu Grunde gelegt werden, nach welchen Eigenschaften der zu typisierenden Objekte also die Klassifikation erfolgt:

- *Struktureigenschaften* – Statische Strukturbestandteile von Komponenten wie Interfaces, Operationen, Nachrichten,
- *Verhaltenseigenschaften* – Dynamische Verhaltensaspekte von Komponenten. Etwa zeitliche bzw. kausale Bedingungen über funktionalen Struktureinheiten.

Bezüglich der im Rahmen einer Komponentenarchitektur bedeutsamen Informationen sind beide Varianten relevant, denn bei einer Komposition von Bausteinen müssen diese sowohl strukturell als auch dynamisch-funktional »zusammenpassen«. Sollen dann also beide Eigenschaften durch eine Typisierung berücksichtigt werden, bestehen zwei Möglichkeiten der Kombination:

- *Strukturelle und dynamische* Eigenschaften in *zwei getrennten Typsystemen*.
- *Strukturelle und dynamische* Eigenschaften, vereint in *einem einzelnen Typsystem*.

Der Ansatz des Autors entspricht dabei der letzteren Variante, indem »klassische« Schnittstelltenbeschreibungen ebenso in die Klassifikation einfließen wie operationale Verhaltensbeschreibungen. Der Vorteil eines einzelnen Typmodells liegt in der direkten Assoziationsmöglichkeit der Teilspezifikationen. Zugleich zwingt dies jedoch zu einer *Strukturierung* des Typmodells, um unterschiedlich qualifizierte Beschreibungselemente aufnehmen und sinnvoll anordnen bzw. wechselseitig zuordnen zu können. Ein solches Typmodell soll nun konkretisiert werden.

4.2 Typmanagement

Es stellt sich zunächst die Frage nach der Definition des *Komponententyps*. Programmiersprachlich ist ein (Daten-) Typ eine Wertemenge, auf der bestimmte Operationen definiert sind. In

Bezug auf Komponenten muss allerdings einerseits die Bedeutung der Komponente (Semantik) für die Gesamtanwendung berücksichtigt werden. Dies betrifft die Frage, was die Komponente hinsichtlich ihrer Anwendungssemantik zu leisten vermag. Andererseits stellt sich die Frage nach ihrer Benutzung im Hinblick auf die programmiersprachlich relevanten Komponentenschnittstellen (Syntax).

Der programmiersprachliche Typbegriff stellt sich als zu ausdrucksschwach dar, sodass sich eine adäquate Typisierung von Softwarekomponenten nicht realisieren lässt, wenn diese lediglich auf der Basis von Datentypen vorgenommen wird. Vielmehr ist es die am fachlichen Problem orientierte Anwendungssemantik, die in den Vordergrund rückt, sodass der Komponententyp im Wesentlichen durch die Eigenschaften des Verhaltens einer Komponente bestimmt werden sollte. Dennoch muss zusätzlich zu dieser Charakterisierung auch ein syntaktischer Typbegriff berücksichtigt werden, da die Komponente letztlich auf technischer Ebene typisierte Schnittstellen besitzt. Der syntaktische Typ kann dabei als Erweiterung des programmiersprachlichen Typbegriffs aufgefasst werden, da er sich aus Datentypen ableiten lässt. Für eine Komponente ist dies der sogenannte *operationale Schnittstellentyp* [Christiansen & Münke, 1996], der sich aus der Schnittstellenbeschreibung ergibt. Zur Beschreibung der Schnittstelle kann dabei eine herkömmliche Interface Definition Language (IDL) verwendet werden.

Nach dieser Trennung zwischen semantischem und syntaktischem Typ muss der Semantik einer Softwarekomponente mehr Bedeutung zugemessen werden, da der syntaktische Typbegriff technisch motiviert, und damit aus dem Blickwinkel einer Anwendungsdomäne eher »künstlich« ist. Dies wird auch dadurch unterstrichen, dass die Komponentenschnittstellen gegebenenfalls adaptiert werden können wie Kapitel 3 gezeigt hat. Eine solche Adaption ist möglicherweise notwendig, wenn erst dadurch eine Interaktion zwischen Komponenten ermöglicht wird. Beide Typbegriffe zusammen erlauben schließlich die Realisierung einer Komponentenklassifikation sowie die Prüfung einer typsicheren (korrekten) Komponenteninteraktion[5], wobei der seman-

[5] Analog zur Typprüfung in Programmiersprachen durch den zu Grunde liegenden Compiler kann vor der Komponentenkopplung eine Typprüfung durch einen Typmanager erfolgen. Eine Typprüfung zur Laufzeit ist mit den Ansätzen der vorliegenden Arbeit insbesondere in Form aktiver Konnektoren denkbar (vgl. a. [Griffel, 1996]), aber nicht explizit vorgesehen, da zum einen ein abgeschlossener Konstruktionsprozess im Vordergrund steht (vgl. Kapitel 5) und zum anderen eine

tische und syntaktische Typ der Komponenten verglichen beziehungsweise deren Kompatibilität geprüft werden kann. Grundvoraussetzung dafür ist allerdings im ersten Schritt die Festlegung des Komponenten-Meta-Modells auf Basis einer interoperablen, technologieunabhängigen Typbeschreibung, die die Eigenschaften des Komponententyps reflektiert und auf deren Konzeption im Folgenden eingegangen wird.

4.3 Konzeption des Typmodells

Der Ansatz dieser Arbeit basiert auf einem Typmodell, das sich aus dem *Interaktionspotential* (Interaktionsmuster) der zu Grunde liegenden Komponenten ergibt, d.h. dem eingehenden und dem ausgehenden Informationsfluss in Form sogenannter *Domänenaktivitäten*. Der Begriff der *Aktivität* innerhalb einer Domäne wurde hier gewählt, um einen direkt innerhalb eines Anwendungsfalls des Problembereichs beobachtbaren Vorgang zu kennzeichnen und somit eine Abgrenzung zu technisch motivierten Begriffen wie Methodenaufrufen etc. zu erhalten. Im Verlauf der weiteren Darstellung insbesondere auch in Kapitel 6 wird der Begriff des *Business Event* (kurz BE) äquivalent zu dem der Domänenaktivität verwendet, da dieser in gleichem Sinne sich häufig in der Literatur findet (vgl. etwa [OMG, 1999] [Herzum & Sims, 2000]).[6] Der Begriff soll hier allerdings keine Einschränkung etwa nur auf die Betrachtung »geschäftlicher« (kommerzieller) Interaktionen implizieren.[7]

Definition 4.1 (Domänenaktivität) *Eine Domänenaktivität bezeichnet einen beobachtbaren problemspezifischen Vorgang im Rahmen der innerhalb einer gewählten Domäne (Anwendungsbereich) auftretenden Ereignisse.*

Definition 4.2 (Interaktion) *Eine Interaktion bezeichnet den wechselseitigen Informationenfluss zwischen zwei kommunizierenden Aktoren einer Domäne im Rahmen der Abwicklung bzw. des Auftretens einer oder mehrerer Domänenaktivitäten.*

Typveränderung nur einem Rollenwechsel (vgl. Abschnitt 4.8) eintritt, der wiederum zum Entwicklungszeitpunkt festgelegt wird.

[6] vgl. a. die Bildung eines Typbegriffs auf Basis geordneter Ereignismengen in Rapide [Allen, 1997]. Der dort verwendetete Begriff des *Port* einer Komponente entspräche am ehesten der hier verwendeten Vorstellung des Business Event Begriffs.

[7] Beispielsweise ließen sich *Geschäftsprozesse*, die auf Basis entsprechend typisierter Komponenten realisiert sind, direkt als Sequenzen von Domänenaktivitäten modellieren.

Das Typmodell ist somit interaktionsbetont, sodass aufgrund auftretender Interaktionsmuster *Kategorien* von Komponenten gebildet werden können, deren *Verhalten* dann über die zu beobachtenden Interaktionmuster definiert ist. Domänenaktivitäten sind dabei nicht als technische Ereignisse aufzufassen. Sie drücken vielmehr Interaktionen auf Modellebene aus, die zunächst lediglich semantisch – im Rahmen einer Anwendungsdomäne – definiert sind. Eine Abbildung der Aktivitäten auf die programmiersprachlich definierte Funktionalität an den Komponentenschnittstellen führt schließlich zur Realisierung ihrer Semantik. Domänenaktivitäten können dann Aktivitäten *innerhalb* der Komponente auslösen sowie deren internen Zustand verändern.

4.3.1 Semantischer Typ

Um das Interaktionsmuster einer Komponente auszudrücken, muss jede Domänenaktivität zunächst in ihrer Semantik definiert und repräsentiert werden[8], sodass sie anschließend einer zu typisierenden Komponente zugeordnet werden kann (im Rahmen eines Domänendesigns, vgl. Abschnitt 5.3). Diese Semantikrepräsentation wird im Sinne einer ausgewählten Anwendungsdomäne mit einer festgelegten, standardisierten Terminologie vorgenommen. Hierzu werden im hier vorgestellten Ansatz sogenannte *Merkmalsdiagramme*[9] [SEI, 1993] eingesetzt, die eine hierarchische Dekomposition von Eigenschaften eines zu typisierenden Objekts erlauben (vgl. a. Abschnitt 5.3). Insbesondere erlauben diese Diagramme dann einen *strukturellen* Vergleich der Repräsentationen, sodass auf eine weitergehende Standardisierung zusätzlicher Terminologien verzichtet werden kann.

Die Motivation dieses Ansatzes für das Typmanagement besteht darin, für die zu modellierende Domäne eine vordefinierte Menge von Merkmalen mit festgelegter Semantik zu Grunde legen zu können. Die Merkmale fungieren dabei als bedeutungstragende Elemente der modellierten »Welt«, die aus einer Domänenanalyse gewonnen werden können und im Anschluss daran *standardisiert* werden müssen. Ein Merkmal *NAME* des Typs `String` stellt dann beispielsweise keine beliebige Zeichenkette mehr dar, sondern eine Zeichenkette, die eben einen Namen

[8] Hierin liegt auch die entscheidende Ergänzung bzw. Abstraktion der Nachrichtensicht des Adaptionskonzepts aus dem vorangegangenen Kapitel.

[9] Merkmalsdiagramme entstammen den Methoden der Feature-Oriented Domain Analysis (FODA).

repräsentiert. Insbesondere bietet die Domänenanalyse in diesem Kontext durch die Bereitstellung technologieunabhängiger Fachkonzepte eine problem- sowie anwendungsnahe Softwareentwicklung. Für die nachfolgende semantische Modellierung der Domänenaktivitäten darf nun ausschließlich die standardisierte *Domänenterminologie* in Form der Merkmale verwendet werden, sodass sich das Interaktionspotential der Komponente direkt aus der Menge der ihr zugeordneten Domänenaktivitäten und deren kausalen Abhängigkeiten untereinander ergibt.

Des Weiteren wird das nach außen wahrnehmbare Komponentenverhalten gemäß des Ansatzes aus Kapitel 3 durch eine Protokollmenge festgelegt. Diese beinhaltet Protokolle über die entsprechenden Domänenaktivitäten, d.h. setzt diese in einen kasusalen Zusammenhang. Jedes Protokoll beschreibt dabei genau ein *Segment* in der Schnittstellensicht einer Komponente (vgl. Kapitel 2).[10] Die Protokolle spezifizieren dabei alle gültigen Sequenzen von Domänenaktivitäten im Sinne einer korrekten und funktionsgerechten Verwendung einer Komponente.

Ferner können auf der semantischen Ebene *Constraints* definiert werden, um typrelevante Regeln auszudrücken. Diese werden insbesondere bei Typvergleichen herangezogen, während ihnen bei Kompatibilitätsprüfungen aus Sicht des Typmanagements keine Bedeutung zukommt.[11] So zeichnet sich beispielsweise der Typ *Kreis* durch zwei Attribute *X-RADIUS* und *Y-RADIUS* aus, deren Gleichheit nun gerade den Typ *Kreis* auszeichnet. Constraints werden dabei ebenso durch Merkmalsdiagramme repräsentiert werden, um später homogene Verarbeitungsmechanismen bereitstellen zu können. Zudem erlaubt diese Repräsentation eine so hohe Flexibität, dass keine Einschränkung auf einen bestimmten Formalismus erfolgen muss, sondern je nach dessen Wahl unterschiedliche Elemente zu seiner Darstellung repräsentiert werden können. Insbesondere können die mittels *Merkmalsstrukturen* ausgedrückten Terme der zusammengeführten Logik aus Kapitel 2 direkt in Form dieser Diagramme repräsentiert werden (vgl. a. Abschnitt 6.4.3 und Anhang I).

Um auch typrelevante Eigenschaften einer Komponente im Sinne vorhandener *Attribute* erfassen zu können, zugleich aber die konzeptionelle Vielfals des Modells zu beschränken, las-

[10] Der Begriff wird in Anlehnung an die Begriffswahl der CORBA-3.0-Spezifikation verendet. Dort wird eine abgrenzbare Teilfunktionalität einer Komponente als Segment bezeichnet [OMG, 1999].

[11] Im Rahmen von Kompatibiltitätsfragen übernehmen die instanzbezogenen *Policies* diese Rolle (vgl. a. Abschnitt 3.10).

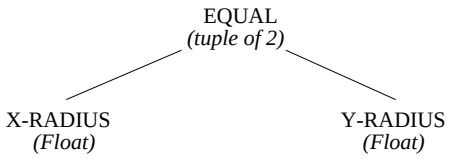


Abbildung 4.1
Merkmalsdiagramm des Kreis-*Constraint*

sen sich Constraints auch als Regeln aufgefassen, die lediglich die Existenz eines bestimmten Komponentenattributs beschreiben, ohne eine zusätzliche Bedingung zu stellen. In diesem Fall wird von einem *Existenz-Constraint* gesprochen. Mit einem solchen könnte beispielsweise das Attribut *Warteschlange* eines Komponententyps *Druckdienst* spezifiziert werden (vgl. a. Abschnitt D.2).

4.3.2 Syntaktischer Typ

Zur Etablierung einer vollständigen Komponententypbeschreibung müssen gemäß des Modells aus Kapitel 2 die syntaktischen Metainformationen über *angebotene* und *verwendete* Schnittstellen spezifiziert werden. Die Domänenaktivitäten materialisieren sich zu diesem Zweck in *Nachrichten* – der Domänenaktivitäts-Syntax, auf deren Basis ein Nachrichtenaustausch zwischen kollaborierenden Komponenten zur Realisierung der gewünschten Anwendungssemantik führt. Diese Nachrichtenebene des Typmodells hat dabei die Aufgabe, eine syntaktische Materialisation des Komponententyps bei einer gleichzeitigen Aufrechterhaltung der technischen Abstraktion. In diesem Sinne wird hier auch davon gesprochen, dass die Nachrichtenebene die Syntax der konkreten (technischen) Komponente *approximiert*. Auf diese Weise kann ein und dieselbe Semantik mit unterschiedlichen Nachrichten(folgen) ausgedrückt werden. Diese Approximation, die ihrerseits als Nachrichtensemantik interpretiert werden kann, wird mittels Mengen von Datentypen realisiert, die die für den Informationsfluss im Rahmen einer Interaktion auszutauschenden Parameter repräsentieren. Analog zur semantischen Ebene der Domänenaktivitäten werden auch auf der Nachrichtenebene Protokolle zur operationalen Beschreibung des Verhaltens eingesetzt.

Letztendlich konkretisiert sich die Nachrichtensemantik auf einer dritten Ebene des Typmodells – der syntaktischen Ebene – in Form von Operationstypen (der Nachrichten-Syntax), die einer signaturbasierten Schnittstellenbeschreibung gleicht. Ferner

wird der *Kommunikationstyp* sowie das zu Grunde liegende *Kommunikationsmodell* festgelegt. Die Abstraktion der Eigenschaften der Kommunikation zwischen Interaktionspartnern trägt zur Erhöhung der Einsatzmöglichkeiten und damit einer verbesserten (Wieder-) Verwendbarkeit einer Komponente bei. Dies ist insbesondere durch die Vielfalt der in *verteilten* Systemumgebungen auftretenden Kommunikationsarten motiviert. Abermals werden auch in dieser Ebene Protokolle zur operationalen Spezifikation des Verhaltens einer Komponente eingesetzt werden.

In der Typbeschreibung einer Komponente verweist dann jede – hier als *semantisch* bezeichnete – Repräsentation einer Domänenaktivität sowie jede Nachrichtensemantik auf genau eine konkrete syntaktische Realisierung, eben die der Komponente auf der untergeordneten Ebene.[12]. Ausgenommen davon sind lediglich die Constraints, da deren semantische Repräsentation keine syntaktische Realisierung zu Grunde liegt. Constraints existieren somit ausschließlich auf der obersten (semantischen) Ebene des Typmodells.

Die oben informal skizzierte Konzeption des Typmodells sowie die Beziehungen seiner einzelnen Elemente soll im Folgenden präzisiert werden.

4.3.3 Formalisierung des Typmodells

Die Typbeschreibung wird im weiteren Verlauf der Arbeit durch das folgende formale Typmodell dargestellt, das in seiner Gesamtheit den Komponententyp repräsentiert und auf dem nachfolgend Typoperationen sowie Funktionen zur Typsuche ausgeführt werden können.

1. Komponententyp
   - hat einen Typnamen (diese muss *nicht* aus einer Standardterminologie gewählt werden und dient lediglich der Refrenzierung durch den menschlichen Betrachter)
   - ergibt sich aus den Mengen der Interaktionen im Rahmen von Domänenaktivitäten (Business Events) sowie deren Protokollmenge,
   - ergibt sich aus der zugeordneten Constraint-Menge (repräsentiert durch Merkmalsdiagramme),
2. Business-Event-Typ

[12] Das Konzept, über die semantische Repräsentation Zugang zur syntaktischen Realisierung zu erhalten, entstammt der Linguistik (lexikalische Semantik [Levi & Rappaport Hovav, 1995]).

(a) semantischer Business-Event-Typ

- ❏ hat einen Typnamen
- ❏ wird repräsentiert durch Merkmalsdiagramm

(b) syntaktischer Business-Event-Typ

- ❏ hat einen Typnamen
- ❏ ergibt sich aus einer Menge von semantischen Nachrichtentypen sowie deren Protokollmenge

3. Nachrichtentyp

(a) semantischer Nachrichtentyp

- ❏ hat einen Typnamen
- ❏ ergibt sich aus der Menge der Datentypen der erforderlichen beziehungsweise zur Verfügung stehenden Parameter des Informationsflusses, insbesondere auch eines Datentyps für zurückgegebene beziehungsweise erwartete Parameter

(b) syntaktischer Nachrichtentyp

- ❏ hat einen Typnamen
- ❏ ergibt sich aus einer Menge von Operationstypen sowie deren Protokollmenge
- ❏ ergibt sich aus dem *Kommunikationstyp* (z.B. synchron, asynchron, disconnected) sowie dem *Kommunikationsmodelltyp* (z.B. Methodenaufruf, Events, Message Queue)[13]

4. Operationstyp

- ❏ hat einen Typnamen (dieser ist analog zu »Methodennamen«, »Event-Namen« etc. zu betrachten)
- ❏ ergibt sich aus der geordneten Aufzählung der Datentypen der erforderlichen beziehungsweise zur Verfügung stehenden Aufrufparameter und eines Datentyps für den zurückgegebenen beziehungsweise erwarteten Rückgabewert

5. Datentyp

- ❏ einfache und zusammengesetzte Datentypen[14]

[13] Zu den Begriffen siehe weiter unten.

[14] Im Rahmen dieser Arbeit wird das Basistypsystem der CORBA-3.0-Spezifikation vorausgesetzt, das sich ohnehin bereits um eine technologische Abstraktion bemüht (s. `www.omg.org`).

Der hierarchische Aufbau des Typmodells ergibt sich an dieser Stelle dadurch, dass sich Typrepräsentationen der übergeordneten Ebene stets auf der darunter liegenden Ebene mehr und mehr konkretisieren bei gleichzeitiger Abnahme des Grades an notwendiger Begriffsstandardisierung. Dies wird durch die Abgrenzung von Semantik (a) und Syntax (b) realisiert, die dabei stets den Übergang zwischen zwei Ebenen ausmacht.

Aufgrund dieser Formalisierung des Typmodells ergibt sich der Typbegriff auf drei Ebenen, wobei auf jeder Ebene Typvergleiche sowie Kompatibilitätsprüfungen angegeben werden können.

- Vergleich des semantischen Typs sowie Prüfung der semantischen Kompatibilität der Komponenten (semantische Ebene)
- Vergleich des approximierten syntaktischen Typs der Komponenten sowie Kompatibilitätsprüfung bezüglich dieser Ebene des Typmodells (Nachrichtenebene)
- Vergleich des syntaktischen Typs sowie Prüfung der syntaktischen Kompatibilität der Komponenten (syntaktische Ebene)

Der Begriff der *semantischen Kompatibilität* betrifft dabei im Sinne des in Kapitel 3 vorgestellten Adaptionskonzepts das »Gegenstück« eines Typs, also einen Komponenttyp mit inversem Interaktionsmuster. Auf der semantischen Ebene kann diese Beziehung durch die emittierten sowie konsumierten Business Events ausgedrückt werden. Auf der Nachrichtenebene sowie der syntaktischen Ebene kann von *syntaktischer Kompatibilität* gesprochen werden, wenn die *Benutzung* der Schnittstelle der Syntax entspricht, die die *angebotene* Schnittstelle festlegt.

Eine wensentliche Motivation dieser Trennung unterschiedlicher Ebenen im Typmodell ist die Kapselung verschiedenartiger Metainformationen zur Optimierung der späteren Verarbeitung und der Ableitung von Aussagen zu Teilaspekten einer Komponente. So umfasst beispielsweise die semantische Ebene ausschließlich Informationen, die für die Anwendungssemantik relevant sind. Ein Typmanagement muss dadurch bei einem semantischen Vergleich nur diese Ebene berücksichtigen. Die Evaluierung von Beziehungen zwischen Komponententypen beginnt dabei stets auf der semantischen Ebene.[15]. Ein weiterer Abgleich

[15] Über die Semantik kann die *konzeptionelle* Übereinstimmung geprüft werden.

auf den untergeordneten Ebenen wird lediglich dann vorgenommen, wenn auf der übergeordneten Ebene eine sinnvolle Beziehung abgeleitet werden konnte. In einem typischen Szenario vergleicht ein Typmanager zunächst die semantischen Komponententypen beziehungsweise prüft deren semantische Kompatibilität. Nur für den Fall semantisch korrelierender Komponenten besteht dann die Notwendigkeit einer weiteren, syntaktischen Analyse. Diese interne Aufspaltung des Typbegriffs erlaubt eine spätere Unterstützung im Sinne eines *progressiven Filterns* [Oberndorf et al., 1997] bei der Auswahlentscheidung einer Komponente durch einen Anwender oder Generator (vgl. Kapitel 5).

Das formalisierte Typmodell kann durch die folgende Baumrepräsentation dargestellt werden, die sich aus den beschriebenen drei Ebenen zusammensetzt. Der *Typbaum* stellt dabei die interne Struktur des Komponententyps dar.[16]

Um später die individuelle *Identität* aller, beim einem Typmanager registrierten Komponenten zu erhalten, wird für jede Komponente ein eigener Typbaum angelegt. Folglich muss bei nachfolgenden Vergleichsoperationen nicht die Referenz, sondern der Wert eines jeden Elements des Typbaums zu Grunde gelegt werden. Ein Typmanager muss demnach die entsprechenden Typbäume *by-value* (Wertsemantik) vergleichen. Alternativ wäre ebenso ein Modell denkbar, in dem der Typmanager bei der Registrierung einer Komponente seine eigene Wissensbasis (vgl. a. Abschnitt 6.2) nur soweit erweitert, dass er ausschließlich die Teilbäume des Typmodells neu hinzufügt, die für ihn *neue* Metainformationen darstellen. Dies hätte dann allerdings den Verlust der Komponentenidentität zur Folge.[17]

Im Rahmen eines Typmanagements erhält jede Komponente dann eine global eindeutige Komponenten- sowie Typkennung (ID), die dem korrespondierenden Typbaum zugeordnet werden. Die Komponenten-ID wird dabei der Instanzebene zugeschrieben. Alle registrierten *Komponenten* unterscheiden sich folglich in ihrer Komponenten-ID, wohingegen die Typ-ID die verschiedenen *Komponententypen* voneinander abgrenzt. Gleiche Typen erhalten somit die gleiche Typ-ID, wobei Gleichheit auf allen drei Ebenen des Typbaums vorausgesetzt wird (vollständige Gleichheit, s.u.).

[16] Die punktierten Linien weisen darauf hin, dass sich unter der obersten Ebene stets 1 bis n Ebenen befinden, nicht nur jeweils eine, wie die Grafik auf den ersten Blick vermuten lassen könnte.

[17] Die Identität einer Komponente soll für Mechanismen wie das Trading oder die Verwaltung eines Komponentenpools erhalten bleiben (vgl. a. Kapitel 5).

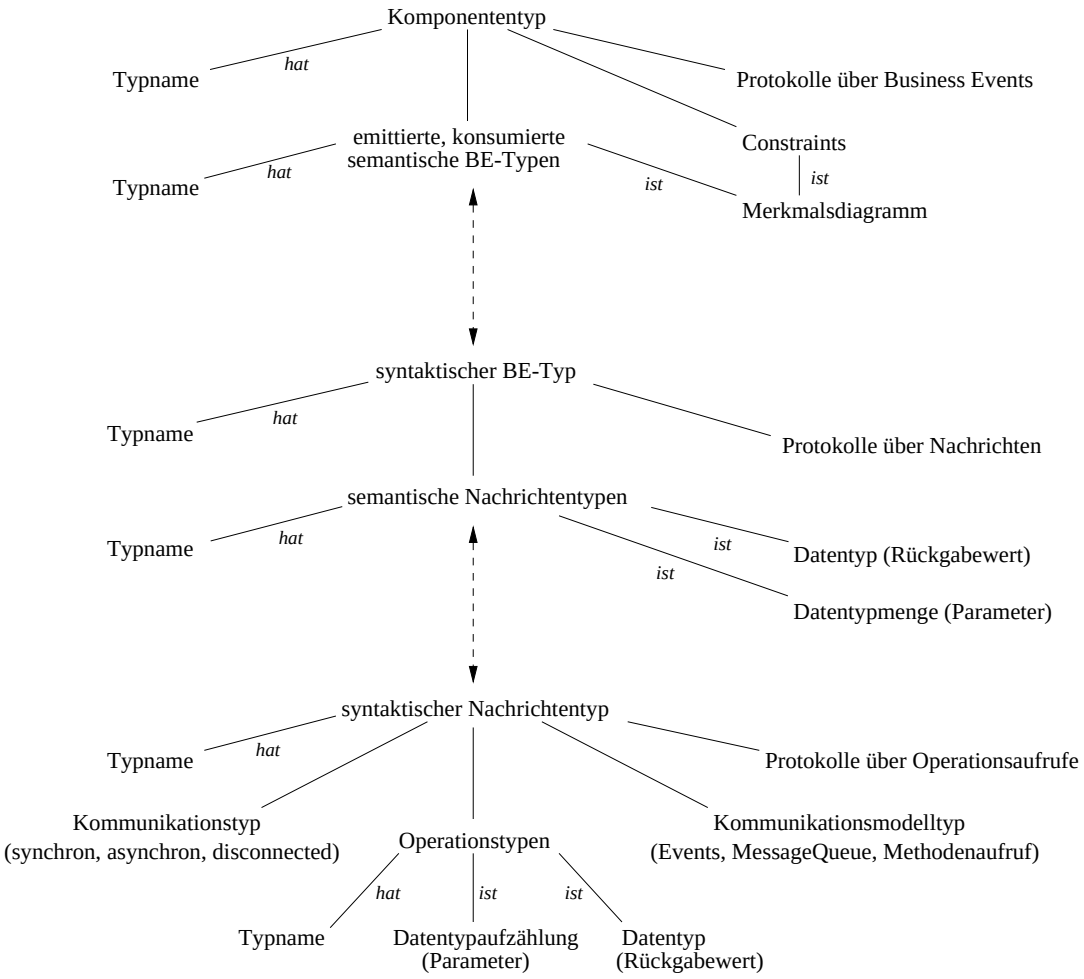


Abbildung 4.2
drei-stufiges Typmodell

4.3.4 Protokollrepräsentation durch endliche Automaten

Die Motivation des Einsatzes von Protokollen auf den drei Ebenen des Typbaums ergibt sich aus den positiven Erfahrungen mit dem vorgestellten Adaptionskonzept und der entsprechenden Notwendigkeit, mögliche Sequenzen von Domänenaktivitäten, Nachrichten und Operationen festzulegen, um ein semantisch reicheres Modell zu erhalten. Protokolle werden dabei entsprechend den in Kapitel 3 ausgearbeiteten Konzepten als endliche Automaten (Finite State Machines) [Hopcroft & Ullman, 1988] dargestellt, wobei die Menge aller gültigen Sequenzen – auch

Sprache des Automats genannt – das Verhalten[18] der Komponente auf der jeweiligen Ebene beschreibt. Ein Typmanager hat somit die Möglichkeit, Komponentenprotokolle zu vergleichen. Dieser Vergleich zielt insbesondere darauf ab, die Gleichheit von Protokollen zu prüfen, um eine Aussage über die Verwendung der von den Protokollen beschriebenen (abstrakten) Schnittstellen (Segmente) zu machen. Für den Fall der Gleichheit kann dann von einem gleichen, gemeinsamen Verständnis der Verwendung ausgegangen werden. Die theoretische Informatik unterscheidet in diesem Zusammenhang zwischen deterministischen und nicht-deterministischen Automaten. Ein Automat wird als deterministisch (DFA) bezeichnet, wenn für jeden Zustand jeder Zustandsübergang eindeutig ist, also in genau einen Folgezustand führt.

4.3.4.1 Eindeutigkeit von Protokollen

Um die Gleichheit von Protokollen zu prüfen, setzen viele effiziente Algorithmen deterministische Automaten voraus. Diese Forderung stellt allerdings kein wesentliches Problem dar, denn es lässt sich zu jedem nicht-deterministischen Automat (NFA) ein äquivalenter DFA konstruieren, der die gleiche Sprache akzeptiert (Potenzautomat). Vielmehr ergibt sich an dieser Stelle eine Problematik, die aus der *Beschreibung* der Zustandsübergänge resultiert.

Die theoretische Informatik definiert Zustandsübergänge auf der Basis von *Zeichenketten*, die entweder gleich oder ungleich sein können. Der Begriff des Determinismus ergibt sich somit in dem Sinne, dass von jedem Zustand höchstens ein Folgezustand mit der zu Grunde liegenden Zeichenkette als Kantenbeschriftung erreichbar ist.

Durch das oben festgelegte Typmodell besteht nun aber die Möglichkeit, Zustandsübergänge *strukturell* zu definieren.[19] Demnach wird nicht der Name der entsprechenden Domänenaktivität, der Nachricht beziehungsweise der Operation für einen Vergleich herangezogen, sondern vielmehr deren strukturelle Beschreibung.[20] Nun können strukturelle Beschreibungen allerdings – im Gegensatz zu simplen Zeichenketten – unter anderem auch in Supertyp- / Subtyp-Beziehungen stehen, was ein Problem

[18] beziehungsweise das Verhalten einzelner Segmente

[19] Dies war ja gerade das Ziel, um eine semantisch ausdruckstarke Repräsentation zu erhalten.

[20] eben in Form von Merkmalsstrukturen (engl. *feature structures*) bzw. den geordneten Mengen ihrer syntaktischen Elemente

für die Aufrechterhaltung des Determinismusbegriffs darstellt, da im Vergleichsautomat somit mehr als eine Entsprechung (gleiche Struktur) zu einem gegebenen Zustandsübergang existieren kann.

Letztendlich muss zur Gewährleistung der Eindeutigkeit von Protokollen gefordert werden, dass – für jeden Zustand innerhalb eines Automats – die zu Grunde liegenden strukturellen Beschreibungen der Zustandsübergänge in *keiner* Beziehung – weder Gleichheit, noch Supertyp / Subtyp – zueinander stehen. Diese Forderung kann in den Automaten durch die Einführung spezieller Kanten – hier als ω-Kanten (»reserviertes« Wort) bezeichnet – umgesetzt werden, die Zustandsübergänge beschreiben, denen keine strukturelle Beschreibung zu Grunde liegt. Folglich sind dann auch die bezüglich einer strukturellen Zustandsübergangsbeschreibung erweiterten endlichen Automaten eindeutig, sodass zu einem gegebenen Zustandsübergang im Vergleichsautomat höchstens eine Entsprechung existiert. Eine Verwendung von ω-Kanten ist an dieser Stelle allerdings nicht unproblematisch (vgl. Anhang F). Ein Algorithmus zur Prüfung der Gleichheit (Äquivalenz) von DFAs, der bezüglich strukturell definierter Zustandsübergänge erweitert ist, findet sich in Abschnitt 6.2.2.1.

4.4 Beziehungen zwischen Typen

4.4.1 Klassifikation und Typisierung

Die klassische Typtheorie definiert Typen als Wertemengen, deren Elemente bestimmte gemeinsame Eigenschaften zu Grunde liegen. Als Subtyp wird danach eine Teilmenge verstanden, die sich einerseits durch die Eigenschaften der Obermenge – der des Supertyps – auszeichnet, die aber andererseits noch weitere, Subtyp-spezifische Eigenschaften festlegt. Folglich umfasst der Subtyp »mehr« als der korrespondierende Supertyp, was zu der Implikation führt, dass ein Supertyp stets durch seine Subtypen substituiert werden kann, sodass diese somit im Kontext des Supertyps verwendet werden können [Cardelli & Wegner, 1985].

Im Komponentenkontext kann dann von einem Subtyp gesprochen werden, wenn ein Komponententyp für jede Verwendung eines anderen Komponententyps – des Supertyps – eingesetzt werden kann [Griffel, 1998]. Für die über die Komponentenschnittstelle *angebotene* Funktionalität ergibt sich dabei der folgende operationale Subtypbegriff.

Eine Funktionsspezifikation F' mit den Parametertypen $q_1, \ldots, q_n$ definiert genau dann einen Subtyp einer Funktionsspezifikation F mit den Parametertypen $p_1, \ldots, p_n$, wenn alle Parametertypen p_i von F Subtypen der Parametertypen q_i von F' darstellen (Kontravarianz) sowie der Rückgabetyp von F' einen Subtyp dessen von F beschreibt (Kovarianz).

Für beide Funktionsspezifikationen wird in diesem Kontext stets die gleiche Parameteranzahl vorausgesetzt. Der Ko- beziehungsweise Kontravarianzbegriff drückt dabei die *Richtung* des Subtypverhältnisses der Parametertypen beziehungsweise des Rückgabetyps im Vergleich zu dem der Funktionstypen aus. Ist dieses Verhältnis gleich, so wird von Kovarianz gesprochen, während ein entgegengesetztes Verhältnis als Kontravarianz bezeichnet wird [Cardelli, 1989].

Der durch die Objektorientierung eingeführte Begriff der Klasse soll im Folgenden dem Typbegriff der klassischen Typtheorie gegenübergestellt werden, um eine Vorstellung eines für die Komponentensicht sinnvollen Typbegriffs zu erhalten.

4.4.1.1 Typ vs. Klasse

Typ und *Klasse* stehen seit Einführung des Klassenbegriffs durch die Objektorientierung in enger Beziehung zueinander. Allerdings muss klar zwischen ihnen differenziert werden, wobei beide Begriffe in der Praxis dennoch häufig verschwimmen, sodass dies oftmals in ihrer synonymen Verwendung resultiert.[21]

Die in der Praxis gängigen objektorientierten Programmiersprachen bieten mit dem Klassenkonzept – abgesehen von primitiven Datentypen wie beispielsweise in Java – zunächst das einzige Ausdrucksmittel einer Typisierung. Die damit einhergehende Vermischung typisierender sowie strukturierender Konzepte führt dabei zu Problemen, wie sie etwa das »Kreis-Ellipse«-Problem verdeutlicht: Während die klassische Typtheorie auf Basis ihrer (Teil-) Mengen-orientierten Sichtweise den Kreis als Subtyp der Ellipse ansehen würde, so könnte – aus der Subklassensicht der Objektorientierung mittels Vererbung – die Ellipse ebenso eine Erweiterung des Kreises darstellen, da diese einen zweiten, *zusätzlichen* Radius definiert. Eine typrelevante Regel (Constraint) kann an dieser Stelle zur Einhaltung der

[21] Sprachen wie ADA zeigen jedoch, dass auch die praktische Programmierung sehr von einer strikten *Trennung* einerseits *strukturierender* und andererseits *klassifizierender* Konstrukte profitieren kann.

klassischen Typsicht beitragen, indem sie als zusätzliche Randbedingung des Kreises die Gleichheit beider Radien fordert. Des Weiteren ergeben sich in diesem Kontext Vererbungsanomalien [Dodani & Tsai, 1992], die eine Abgrenzung von Typ und Klasse notwendig erscheinen lassen. Die Komponentensicht verzichtet unter anderem aus diesem Grund auf Vererbungsbeziehungen zwischen einzelnen Softwarebausteinen.

Ein wesentlicher Unterschied lässt sich insbesondere an den Begriffen des *Subtyps* sowie der *Subklasse* aufzeigen, da der Subtypbegriff die Substitutionsfähigkeit der Typausprägung im Hinblick auf eine klar definierte Austausch- beziehungsweise Ersetzbarkeit in den Vordergrund stellt, während das Subklassenkonzept den Schwerpunkt auf die gemeinsame Nutzung von Implementationscode legt. Dabei ist die Gewährleistung der Substitutionseigenschaft – wie sie der Subtyp stets aufrecht erhält – für die Subklasse nicht unbedingt gegeben, da zum Beispiel ein Überschreiben von Methoden der Superklasse zu einem anderen Verhalten der Subklasse führen kann. Dennoch kann eine Korrelation zwischen Subtyp und Subklasse postuliert werden, wie sie etwa in [Abadi & Cardelli, 1996] abgeleitet wird.

Eine Objektinstanz der Klasse c' ist ein Subtyp einer Objektinstanz der Klasse c, wenn c' Subklasse von c ist sowie alle von c' überschriebenen Methoden Methodenspezialisierungen (Subtypen) derer von c sind.

Diese Korrelation stellt somit hinsichtlich der Differenzen von (Sub-) Typ und (Sub-) Klasse heraus, dass für überschriebene Methoden der Superklasse die Typsicherheit zu Grunde liegender Signaturen sichergestellt sein muss. In [Abadi & Cardelli, 1996] wird dies als Methodenspezialisierung bezeichnet, die durch kontravariante Parametertypen mit kovariantem Rückgabetyp auf der Seite der Subklasse realisiert wird und dem bereits dargestellten operationalen Subtypbegriff zwischen Funktionsspezifikationen entspricht. Verhaltenskonformität kann dabei allerdings nicht gewährleistet werden, da eine überschriebene Methode der Superklasse möglicherweise interne Zustände der Klasse anders ausnutzt, als dies unter Umständen deren Implementierung auf Seiten der Subklasse vorsieht. Konform ist somit lediglich die Zuweisung des Subtyps an die Instanz des korrespondierenden Supertyps, die durch die Prinzipien der Ko- und Kontravarianz sichergestellt wird.[22]

[22] Eine entsprechende Diskussion der Schwäche der Varianzbegriffe findet sich etwa auch in [Griffel, 1998].

Wie bereits im Rahmen der Konzeption eines Komponentemodells in Kapitel 2 erörtert, sind in diesem Zusammenhang Ansätze zu berücksichtigen, die eine Verhaltensspezifikation ermöglichen und dadurch ein (semantisch) reicheres Modell ausmachen. Dabei stellt sich die Frage, ob eine Verhaltensbeschreibung eher dem Klassenkonzept der Objektorientierung, das bereits Implementationsaspekte umfasst, oder vielmehr dem abstrakteren Konzept des Typs zugeordnet werden soll. Das dieser Arbeit zu Grunde liegende Typmodell zeigt, dass eine derartige Verhaltensspezifikation bereits auf einer übergeordneten abstrakten und von der konkreten Programmiertechnik losgelösten Ebene vorgenommen werden kann. Dies ist insbesondere durch [Liskov & Wing, 1993] mit der Definition eines verhaltensbasierten Subtypbegriffs motiviert[23]. Somit kann dann im Komponentenkontext eine adäquate Behandlung von Konformität und Kompatibilität gewährleistet werden.

4.4.1.2 Beziehungen im Typmodell

Die bisherigen Überlegungen können an dieser Stelle auf die potenziellen Beziehungen im Typmodell übertragen werden, wobei deren Baumrepräsentation (Typbaum) die interne Struktur des Komponententyps widerspiegelt.

Als eine charakteristische Beziehung zwischen Mengen sollen nachfolgend die Begriffe der *Erweiterung* sowie der *Reduzierung* verwendet werden, wenn eine Menge mehr beziehungsweise weniger Elemente als eine Vergleichsmenge umfasst. Folglich sind diese Beziehungen für alle im festgelegten Typmodell definierten Mengen relevant – den Protokollmengen, der Constraint-Menge, den Domänenaktivitäts-, Nachrichten- sowie Operationsmengen. Im Hinblick auf die Ersetzbarkeit muss ein Substitutionskandidat somit mindestens der jeweiligen Basismenge Rechnung tragen, wobei er diese gegebenenfalls erweitern kann.

Der operationale Aspekt wird durch die Definition der semantischen Nachrichten- sowie Operationstypen im Typmodell abgedeckt. Als relevante Charakteristik erscheinen dabei die Beziehungen zwischen den Datentypen der Parameter sowie den Rückgabetypen, sodass sich in diesem Fall der Subtypbegriff für Funktionsspezifikationen als signifikant herausstellt. Durch diese operationale Begriffsdefinition kann somit die Substitutionseigenschaft sowie eine typsichere Interaktion zwischen Komponenten auf technischer Ebene geprüft werden, indem gemäß den

[23] Dieser stützt sich allerdings ausschließlich auf die Einführung von Vor- und Nachbedingungen (vgl. a. Kapitel 2)

Konzepten aus Kapitel 3 zusätzlich zur Betrachtung der *angebotenen* Funktionalität auch die *benötigte* Funktionalität berücksichtigt werden kann, die ebenso operational beschrieben ist.

Neben dem Subtyp beziehungsweise dem Supertyp lassen sich im Folgenden weitere Typausprägungen definieren, die auf Basis des festgelegten Typmodells entstehen können.[24] Zur Etablierung einer einheitlichen Terminologie soll nachfolgend der Begriff der *Typart* verwendet werden, der eine gewisse Nähe (Affinität) der entsprechenden Typausprägung zu einem Basistyp ausdrückt. Im Anschluss daran liegt der Schwerpunkt auf der Betrachtung der Konformität der verschiedenen Typarten zu ihrem Basistyp, wobei wiederum die Frage nach der Substitutionsfähigkeit im Mittelpunkt steht. Für die verschiedenen Typarten werden sich im Hinblick auf diese zentrale Fragestellung unterschiedliche Aussagen ergeben.

4.4.2 Typarten

Typarten können auf allen drei Ebenen des Typbaums definiert werden. Der *Subtyp* macht an dieser Stelle die erste festzulegende Typart aus, die im weiteren Verlauf auch als *Typart-1* bezeichnet wird und dabei die Substitutionseigenschaft inne hat, sodass die korrespondierende Typausprägung stets im Kontext des Basistyps verwendet werden kann[25]. Eine weitere Typart – die *Typart-2* – ergibt sich auf der Basis der Verhaltensbeschreibung, die durch Protokolle und Constraints dargestellt wird. Diese Typvariante beschreibt einen *Verhaltenstyp*, durch den unterschiedliches Verhalten von Komponenten zum Ausdruck gebracht wird. Letztendlich muss noch eine dritte Typart die *Typart-3* – zu Grunde gelegt werden, die ausschließlich auf der syntaktischen Ebene definiert werden kann. Durch sie wird ein *technischer Typ* ausgedrückt, der Unterschiede bezüglich des Kommunikationstyps und / oder des Kommunikationsmodelltyps widerspiegelt. Des Weiteren ergibt sich die Möglichkeit, Typarten untereinander zu kombinieren.[26]. Daraus resultieren die folgenden Kombinationsmöglichkeiten.

[24] Damit stellt das Typmodell im Rahmen eines Typmanagments sehr flexible Typbegriffe bzw. Klassifikationsmöglichkeiten bereit.

[25] Dies impliziert Verhaltensgleichheit.

[26] Kombination bedeutet hier, dass eine Typausprägung die Eigenschaften mehrerer Typarten *gleichzeitig* umfasst.

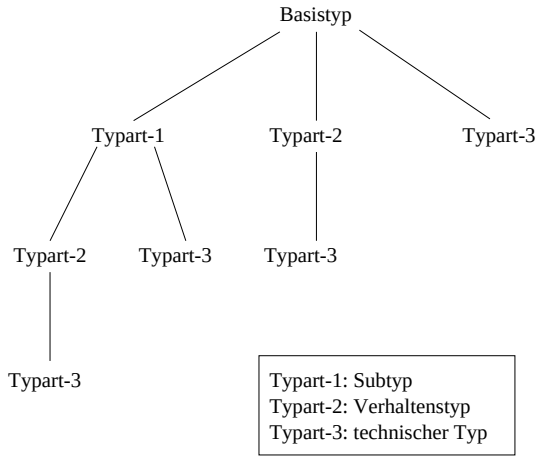


Abbildung 4.3
Typarten

4.4.2.1 Affinität der Typarten zum Basistyp

Typarten unterscheiden sich durch ihre *Affinität*, also ihrer Nähe zum Basistyp, wobei sich durch die definierten Typarten drei Affinitätsebenen ergeben.

Die Typart-3 liegt dabei am nächsten am Basistyp, da in diesem Fall lediglich ein Unterschied auf syntaktischer Ebene hinsichtlich des Kommunikationstyps und / oder des Kommunikationsmodelltyps besteht, gefolgt von der Typart-1, die auf Grund ihrer Substitutionsfähigkeit auch noch relativ nah am Basistyp angesiedelt ist. Die größte Distanz zum Basistyp wird durch eine Beziehung der Typart-2 ausgedrückt. Während sich die Typart-1 auf gleiches Komponentenverhalten stützt ausgedrückt durch die entsprechende Protokollmenge, so basiert die Typart-2 auf einer vom Basistyp abweichenden Verhaltensweise. Die Substitutionseigenschaft kann somit nicht mehr aufrecht erhalten werden.

Tabelle 4.1
Affinitätsebenen

Affinität:	Typart:
1	Typart-3 (`typekind-3`)
2	Typart-1 (`typekind-1`)
3	Typart-2 (`typekind-2`)

Die Gemeinsamkeit aller Typarten besteht schließlich in der impliziten Festlegung ihres eigenen Adaptionsmechanismus (kurz: Adapters), um gegebenenfalls (doch) im Kontext des Basistyps eingesetzt werden zu können. Adapter werden dabei im We-

sentlichen für die Nachrichtenebene sowie die syntaktische Ebene benötigt. Auf der semantischen Ebene wäre gegebenenfalls eine Protokolladaption denkbar.[27] Ein Typmanager hat dann die zusätzliche Aufgabe der Verwaltung adäquater Adaptionsmechanismen, um auftretende Inkompatibilitäten auf den entsprechenden Ebenen des Typbaums beheben zu können. Dazu können explizit als Adapter gekennzeichnete Komponenten registriert werden, die ein Typmanager bei Bedarf automatisch zuordnet. Implizit festgelegte Adaptionsmechanismen lassen sich dabei für jede Typart angegeben.

- Typart-1: Schnittstellenadaption (Anpassung von Operationsnamen)
- Typart-2: Protokolladaption (Anpassung von Aktivitäts-, Nachrichten- und Operationssequenzen)
- Typart-3: Synchronisation, Kommunikationsmodelladaption

An dieser Stelle bleibt festzuhalten, dass die Gleichheit von Protokollen eine notwendige, aber noch keine hinreichende Bedingung für die Substitutionsfähigkeit der Typausprägung darstellt. Potenzielle Substitutionskandidaten sind somit ausschließlich Komponententypen der ersten und zweiten Affinitätsebene. Folglich können auf diesen Affinitätsebenen Adapter zur Überbrückung syntaktischer Inkompatibilitäten *automatisch generiert* (Typart-1, Schnittstellenadaption) beziehungsweise als *Standardadapter* zur Verfügung gestellt werden (Typart-3, zum Beispiel Synchronisation durch aktive Konnektoren). Der Kompatibilitätsbegriff kann dadurch in dem Sinne definiert werden, dass nicht nur ohne Adapter interaktionsfähige Komponenten als kompatibel angesehen werden, sondern vielmehr auch diejenigen, die mittels eines implizit festgelegten, automatisch generierbaren Adapters beziehungsweise vorliegenden Standarddapters koppelbar sind.

Im Gegensatz dazu kann auf der dritten Affinitätsebene für Protokolle nicht ohne weiteres festgestellt werden, in welchem Maße diese voneinander abweichen, sodass unter Umständen kein Adaptionsmechanismus zur Überbrückung der Inkompatibilitäten existiert. Protokolladaption ist daher nicht vollständig automatisierbar, aber es kann zumindest der in Kapitel 3 vorgestellte Halbautomatismus genutzt werden. Eine gewisse Nähe

[27] Dies würde aber die Verarbeitung eines Domänenwissens bedingen, beispielsweise im Sinne eines Expertensystems.

zum Basistyp wird schließlich durch die Beziehungen zwischen den Mengen der Domänenaktivitäten, Nachrichten sowie Operationen beider Komponententypen aufrecht erhalten, sodass die Existenz dieser Affinitätsebene ebenso gerechtfertigt ist.

Komponententypen werden durch Typbäume beschrieben beziehungsweise repräsentiert. Aus dem Vergleich der Werte von Elementen zweier Typbäume lassen sich bestimmte Beziehungen (Assoziationen) ableiten, auf deren Basis anschließend Funktionen zur Typsuche etabliert werden können. Dabei werden wiederum die semantische Ebene, die Nachrichtenebene sowie die syntaktische Ebene separat betrachtet.

4.4.3 Assoziationen im Typmodell

Ausgehend von den verschiedenen Elementen des Typbaums können Assoziationen zwischen Komponententypen auf den drei Ebenen definiert werden. Für die Wurzel-Elemente (Root) dieser Ebenen – den Komponententyp, den syntaktischen Business-Event-Typ sowie den syntaktischen Nachrichtentyp ist die Notwendigkeit der Einführung von *Typarten* (`typekinds`) gegeben, um eine Klassifikation von Typausprägungen zu etablieren. Zu jeder Typart-Beziehung existiert eine korrespondierende komplementäre Beziehung, die den *Basistyp* der Typvariante referenziert. Im Folgenden sind dabei primär die für die einzelnen Elemente des Typbaums festgelegten Assoziationen relevant. Des Weiteren werden für die den Wurzel-Elementen untergeordneten Elemente des Typbaums Ableitungsregeln definiert, auf deren Basis nachfolgend Assoziationen evaluiert werden können. Die Auswertung dieser Assoziationen der Subelemente zur Ableitung einer Aussage für das Wurzel-Element wird im Anschluss daran im nächsten Abschnitt behandelt.

Vorweg anzumerken bleibt noch, dass für die nicht-kommutativen Relationen der Subtypbeziehung `subtype`[28], der Erweiterung `extension` sowie der Typart-1-Beziehung `typekind-1`[29] die komplementären Beziehungen `supertype`, `reduction` und `basetype-1` aus Gründen der Übersichtlichkeit im Folgenden nicht angegeben werden (s. Anhang G). Aus der kommutativen Typart-2- (`typekind-2`) sowie Typart-3-Relation (`typekind-3`) folgt schließlich, dass die korrespondierenden Typausprägungen gleichzeitig auch in der Basistyp-2- (`basetype-2`) beziehungsweise Basistyp-3-Beziehung (`basetype-3`) stehen. Des Weiteren stellen

[28] für untergeordnete Elemente der einzelnen Ebenen des Typmodells
[29] für Wurzel-Elemente der einzelnen Ebenen des Typmodells

Gleichheits- (`equal`), Kompatibilitäts- (`compatibleType`) sowie die speziellen `need-adapter-for`-Assoziationen ebenso kommutative Relationen dar.

4.4.3.1 Semantische Ebene

Die semantische Ebene erlaubt folgende Assoziationen im Typmodell:

Komponententyp:	`equal, typekind-1, typekind-2, typekind-2of-1`[30]`, compatibleType need-adapter-for`
Mengen semantischer BE-Typen:	`equal, extension`
semantischer BE-Typ:	`equal`
Constraint-Menge:	`equal, extension`
Constraint:	`equal`
Protokollmenge:	`equal, extension`
Protokoll:	`equal`

Tabelle 4.2 *Assoziationen auf semantischer Ebene*

Für den Komponententyp ergeben sich Assoziationen zu typgleichen Komponenten sowie Komponententypen, die in den angegebenen Typart-Beziehungen stehen. Typarten können dabei kombiniert werden. Zudem werden zwischen Komponententypen Kompatibilitäts-Assoziationen definiert sowie eine weitere Assoziation, die die Notwendigkeit einer Adaption ausdrückt.

Semantische Business-Event- bzw. Aktivitäts-Typen und Constraints, jeweils dargestellt als Merkmalsdiagramm, werden auf dieser Ebene stets auf *strukturelle* Gleichheit[31] (der Merkmalsdiagramme) geprüft, sodass eine Erweiterung erst zwischen deren *Mengen* etabliert werden kann, wenn eine Menge alle Elemente der Basismenge beinhaltet, aber noch weitere definiert. Analog gilt dies für Protokolle, die sich ebenso nur durch Gleichheit beziehungsweise Ungleichheit auszeichnen, sodass auch hier eine Erweiterung erst zwischen Protokollmengen relevant wird.

[30] vgl. Abb. 4.3

[31] Merkmalsdiagramme lassen sich auf Merkmalsstrukturen (Feature Structures, [Carpenter, 1992]) abbilden, die im Falle gegenseitiger Subsummierung isomorph sind.

4.4.3.2 Nachrichtenebene

Die Nachrichtenebene legt die folgenden potenziellen Assoziationen im Typmodell fest:

syntaktischer BE-Typ:	`equal, typekind-1, typekind-2, typekind-2of-1, compatibleType need-adapter-for`
Menge sem. Nachrichtentypen:	`equal, extension`
semantischer Nachrichtentyp:	`equal, subtype, compatibleType`
Protokollmenge:	`equal, extension`
Protokoll:	`equal`

Tabelle 4.3
Assoziationen auf Nachrichtenebene

Der syntaktische Business-Event- bzw. Aktivitäts-Typ erlaubt Assoziationen zu typgleichen Elementen sowie syntaktischen Business-Event-Typen, die in den festgelegten Typart-Beziehungen stehen. Typarten lassen sich dabei kombinieren. Ferner definieren syntaktische Business-Event-Typen Kompatibilitäts-Assoziationen wie auch eine Assoziation, die die Notwendigkeit einer Adaption spezifiziert.

Semantische Nachrichtentypen definieren auf dieser Ebene Gleichheits-, Subtyp- sowie Kompatibilitäts-Assoziationen. Die Gleichheits- und die Kompatibilitäts-Assoziation sind dabei als kommutative Beziehungen einzustufen, während die Subtyp-Assoziation eine nicht-kommutative Relation darstellt.

Zwei semantische Nachrichtentypen i und j sind ***typgleich****, wenn sie typgleiche Parameter sowie einen typgleichen Rückgabewert bei gleicher Parameteranzahl definieren.*

Ein semantischer Nachrichtentyp j ist ein ***Subtyp*** *eines semantischen Nachrichtentyps i, wenn für alle Parametertypen von i Supertyp-Entsprechungen bei j existieren (Kontravarianz) sowie der Rückgabetyp von j einen Subtyp dessen von i repräsentiert (Kovarianz).*

Zwei semantische Nachrichtentypen i und j sind ***kompatibel****, wenn der dem konsumierten Business-Event zu Grunde liegende Nachrichtentyp einen Subtyp des zu dem emittierten Business-Event korrespondierenden Nachrichtentyps definiert.*

Eine Erweiterung zwischen Mengen semantischer Nach-

richtentypen kommt wiederum zustande, wenn eine Menge zu jedem Element der Basismenge eine typgleiche oder Supertyp- / Subtyp-Entsprechung spezifiziert, aber noch weitere Nachrichtentypen definiert. Protokolle zeichnen sich auch auf dieser Ebene durch Gleichheit beziehungsweise Ungleichheit aus, sodass eine Erweiterung wiederum erst zwischen Protokollmengen betrachtet werden kann.

Den gleichen Assoziationen unterliegt im Folgenden ebenso die syntaktische Ebene, für die nun allerdings ebenso der Kommunikationstyp wie auch der Kommunikationsmodelltyp sowie die damit einhergehende Typart-3 betrachtet werden muss.

4.4.3.3 Syntaktische Ebene

Auf der syntaktischen Ebene sind folgende Assoziationen im Typmodell vorgesehen:

syntaktischer Nachrichtentyp:	`equal`, `typekind-1`, `typekind-2`, `typekind-3`, `typekind-2of-1`, `typekind-3of-1`, `typekind-3of-2`, `typekind-3of-2of-1` `compatibleType`, `need-adapter-for`
Menge von Operationstypen:	`equal`, `extension`
Operationstyp:	`equal`, `subtype`, `compatibleType`
Protokollmenge:	`equal`, `extension`
Protokoll:	`equal`
Kommunikationstyp:	`equal`
Kommunikationsmodelltyp:	`equal`

Tabelle 4.4
Assoziationen auf syntaktischer Ebene

Für den syntaktischen Nachrichtentyp ergeben sich Assoziationen zu typgleichen Elementen sowie syntaktischen Nachrichtentypen, die in den angegebenen Typart-Beziehungen stehen. Typarten können wiederum kombiniert werden. Zudem werden zwischen syntaktischen Nachrichtentypen Kompatibilitäts-Assoziationen definiert sowie eine weitere Assoziation, die die Notwendigkeit einer Adaption ausdrückt.

Operationstypen erlauben auf dieser Ebene Gleichheits-, Subtyp- sowie Kompatibilitäts-Assoziationen. Die Gleichheits- wie auch die Kompatibilitäts-Assoziation sind abermals kommutative Beziehungen, während die Subtyp-Assoziation eine nicht-kommutative Relation darstellt.

Zwei Operationstypen i mit den Parametertypen $p_1, \ldots, p_n$ und j mit den Parametertypen $q_1, \ldots, q_n$ sind ***typgleich****, wenn zwischen allen Parametertypen p_i und q_i Typgleichheit besteht sowie typgleiche Rückgabewerte definiert sind.*

Ein Operationstyp j mit den Parametertypen $q_1, \ldots, q_n$ ist ein ***Subtyp*** *eines Operationstyps i mit den Parametertypen $p_1, \ldots, p_n$, wenn für alle Parametertypen q_i eine Subtyp-Entsprechung p_i existiert (Kontravarianz) sowie der Rückgabetyp von j einen Subtyp dessen von i repräsentiert (Kovarianz).*

Zwei Operationstypen i und j sind ***kompatibel****, wenn der dem konsumierten Business-Event zu Grunde liegende Operationstyp einen Subtyp des zu dem emittierten Business-Event korrespondierenden Operationstyps definiert.*

Eine Erweiterung zwischen Mengen von Operationstypen lässt sich wiederum ableiten, wenn eine Menge zu jedem Element der Basismenge eine typgleiche oder Supertyp- / Subtyp-Entsprechung spezifiziert, aber noch weitere Operationstypen definiert. Protokolle zeichnen sich auch auf dieser Ebene durch Gleichheit beziehungsweise Ungleichheit aus, sodass eine Erweiterung abermals erst zwischen Protokollmengen relevant wird. Letztendlich wird für den Kommunikationstyp sowie den Kommunikationsmodelltyp eine Assoziation definiert, die die Gleichheit – im Falle gleicher Attributwerte – ausdrückt.

4.4.3.4 Benutzerdefinierte Assoziationen

Für die Wurzel-Elemente der drei Ebenen kann der Benutzer eines entsprechenden Typmanagements *manuell* Assoziationen zwischen Typbäumen festlegen. Die Motivation dieser benutzerdefinierten Relationen besteht dabei in der Möglichkeit, für bestimmte Wurzel-Elemente zweier Typbäume explizit anzugeben, dass diese durch einen Adapter zu überbrücken sind, der noch entwickelt sowie registriert werden muss. Dies kann beispielsweise dann der Fall sein, wenn eine Aussage über den benötigten Adaptionsmechanismus nicht automatisch abgeleitet werden kann (vgl. Kapitel 3. Diese Beziehung zwischen Wurzel-Elementen wird durch die Assoziation `need-adapter-for` ausgedrückt, die nicht in die folgende Auswertung der Assoziationen im Typmodell eingeht.

4.5 Assoziationsauswertung im Typmodell

Um Beziehungen zwischen Typbäumen bezüglich ihrer Wurzel-Elemente auf den drei Ebenen abzuleiten, müssen Assoziationen zwischen deren Subelementen ausgewertet werden. Für jede einzelne Ebene werden Assoziationen lediglich dann berechnet, wenn auf der übergeordneten Ebene eine Beziehung abgeleitet werden konnte. Zudem werden auf den einzelnen Ebenen des Typbaums ausschließlich Elemente verglichen, deren (semantische) Repräsentationen auf der übergeordneten Ebene bereits einander zugeordnet wurden. Durch diesen Filtermechanismus kann der Suchraum beziehungsweise die potenzielle Anzahl von Kombinationsmöglichkeiten auf den untergeordneten Ebenen stark eingeschränkt werden. Die folgenden Übersichten beinhalten die möglichen Vergleichsaussagen für den Komponententyp, den syntaktischen Aktivitäts-Typ (Business-Event) sowie den syntaktischen Nachrichtentyp in tabellarischer Form.

4.5.1 Semantische Ebene

Auf der semantischen Ebene lassen sich folgende Vergleichsaussagen für den Komponententyp ableiten:

BE-Mengen	Constraint-Menge	Protokollmenge	Komponententyp
`equal`	`equal`	`equal`	`equal`
`equal/extension`	`equal/extension`	`equal/extension`	`typekind-1`
`equal`	`equal`	`-`	`typekind-2`
`equal/extension`	`equal/extension`	`-`	`typekind-2of-1`

Tabelle 4.5
Vergleichsaussagen für den Komponententyp

In den nachfolgenden Definitionen sind die Elemente der Business-Event-Mengen (BEs) semantische Business-Event-Typen, die einander genau dann entsprechen, wenn sie sich auf strukturgleiche Merkmalsdiagramme stützen (Typgleichheit).

4.5.1.1 Definition Typgleichheit

Zwei Komponententypen i und j sind typgleich, wenn sie die gleiche Protokollmenge sowie Constraint-Menge spezifizieren und gilt:

$$(BEs(emits)_i = BEs(emits)_j) \wedge (BEs(consumes)_i = BEs(consumes)_j)$$

Der formale Teil der Definition gewährleistet dabei, dass die beiden Komponententypen auf dem gleichen Interaktionsmuster basieren, ausgedrückt über emittierte beziehungsweise konsumierte Domänenaktivitäten (Business-Events).

4.5.1.2 Definition Typart-1 (Subtyp)

Ein Komponententyp j stellt eine Typart-1 eines Komponententyps i dar, wenn j eine gleiche oder erweiterte Protokollmenge sowie Constraint-Menge von i spezifiziert und gilt:

$$(BEs(emits)_i \subseteq BEs(emits)_j) \wedge (BEs(consumes)_i \subseteq BEs(consumes)_j)$$

An dieser Stelle gilt somit für eine Typausprägung der Typart-1, dass diese mindestens das Interaktionspotential (in Form eingehender sowie ausgehender Aktivitäten) des Basistyps besitzt, ausgedrückt über die Teilmengenbeziehung der emittierten sowie konsumierten Business-Events. Für den Fall einer solchen *Erweiterung* des Interaktionspotentials des Basistyps ist der Subtyp stets als eine erweiterte Typausprägung zu verstehen, die dadurch eben »mehr« umfasst als der korrespondierende Supertyp und somit die Substitutionseigenschaft aufrecht erhält.

4.5.1.3 Definition Typart-2 (Verhaltenstyp)

Zwei Komponententypen i und j stehen in einer Typart-2-Beziehung, wenn die Protokollmengen von i und j in keiner Beziehung zueinander stehen, beide Komponententypen aber die gleiche Constraint-Menge spezifizieren und gilt:

$$(BEs(emits)_i = BEs(emits)_j) \wedge (BEs(consumes)_i = BEs(consumes)_j)$$

Die formale Definition stellt auch hier wiederum sicher, dass die beiden Komponententypen das gleiche Interaktionsmuster unterstützen.

4.5.1.4 Definition Typart-2 von Typart-1

Ein Komponententyp j stellt eine Typart-2 einer Typart-1 eines Komponententyps i dar, wenn die Protokollmengen von i und j in keiner Beziehung zueinander stehen, aber j eine gleiche oder erweiterte Constraint-Menge von i spezifiziert und gilt:

$$(BEs(emits)_i \subseteq BEs(emits)_j) \wedge (BEs(consumes)_i \subseteq BEs(consumes)_j)$$

Analog zur Definition der Typart-1 ist auch hier mindestens das Interaktionspotential (eingehender sowie ausgehender Aktivitäten) des Basistyps vorhanden (Teilmengenbeziehung).

4.5.1.5 Definition Kompatibilität

Zwei Komponententypen i und j sind kompatibel, wenn sie mindestens ein[32] *gleiches, gemeinsames Protokoll spezifizieren und bezüglich der zu diesem Protokoll korrespondierenden Business-Event-Teilmengen gilt:*

$$\forall x : (x \in BEs(emits)_i \leftrightarrow x \in BEs(consumes)_j)$$
$$\vee$$
$$(x \in BEs(emits)_j \leftrightarrow x \in BEs(consumes)_i)$$

Die formale Bedingung der Kompatibilitätsdefinition stellt in diesem Kontext sicher, dass alle emittierten Domänenaktivitäten vom Interaktionspartner konsumiert werden sowie dass alle konsumierbaren Aktivitäten auch ebenso emittiert werden (vgl. a. den unspezifizierten Empfang in Abschnitt 3.4.5). Die (abstrakte) Schnittstelle wird genau dann von beiden Komponenten vollständig genutzt, sodass von einem gleichen, gemeinsamen Verständnis der Verwendung hinsichtlich der Ausdrucksmächtigkeit der zu Grunde liegenden Protokolle ausgegangen werden kann.

In Analogie zur Ableitung von Aussagen über den semantischen Typ können ebenso Aussagen über den approximierten syntaktischen Typ auf der Nachrichtenebene abgeleitet werden.

4.5.2 Nachrichtenebene

Auf der Nachrichtenebene ergeben sich folgende Vergleichsaussagen für den syntaktischen Business-Event-Typ:

Nachrichtenmenge	Protokollmenge	syntaktischer BE-Typ
`equal`	`equal`	`equal`
`equal/extension`	`equal/extension`	`typekind-1`
`equal`	`-`	`typekind-2`
`equal/extension`	`-`	`typekind-2of-1`

Tabelle 4.6 *Vergleichsaussagen für den syntaktischen Business-Event-Typ*

In den folgenden Definitionen sind die Elemente der Nachrichtenmengen (Messages) semantische Nachrichtentypen, für

[32] Dann kann nämlich wenigstens die (Teil-) Funktionalität eines Segments genutzt werden.

die jeweils typgleiche Elemente oder Supertyp- / Subtyp-Entsprechungen gesucht werden.

4.5.2.1 Definition Typgleichheit

Zwei syntaktische Business-Event-Typen i und j sind typgleich, wenn sie die gleiche Protokollmenge spezifizieren und gilt:

$$Messages_i = Messages_j, \text{ wobei}$$

Typgleichheit für alle einander entsprechenden semantischen Nachrichtentypen gegeben sein muss.

Der formale Teil der Definition stellt dabei sicher, dass sich die beiden syntaktischen Business-Event-Typen aus den gleichen semantischen Nachrichtentypen zusammensetzen.

4.5.2.2 Definition Typart-1 (Subtyp)

Ein syntaktischer Business-Event-Typ j stellt eine Typart-1 eines syntaktischen Business-Event-Typs i dar, wenn j eine gleiche oder erweiterte Protokollmenge von i spezifiziert und gilt:

$$Messages_i \subseteq Messages_j, \text{ wobei}$$

a) im Fall eines ***konsumierten*** *Business Events für jeden semantischen Nachrichtentyp von i eine Subtyp-Entsprechung bei j existieren muss (Kovarianz).*

b) im Fall eines ***emittierten*** *Business Events für jeden semantischen Nachrichtentyp von i eine Supertyp-Entsprechung bei j existieren muss (Kontravarianz).*

Die Typart-1 beinhaltet somit zumindest die semantischen Nachrichtentypen des Basistyps, sodass auch in diesem Kontext für den Fall einer Erweiterung der Nachrichtenmenge der Subtyp eben »mehr« als der korrespondierende Supertyp umfasst und damit eine erweiterte Typausprägung darstellt. Allerdings muss für die Elemente der Nachrichtenmengen eine Fallunterscheidung zwischen konsumierten und emittierten Business-Events vorgenommen werden, wobei dann neben dem Begriff der Erweiterung eine weitere relevante Charakteristik des Subtypbegriffs in den Vordergrund tritt – die *Spezialisierung* beziehungsweise *Generalisierung*.

Da konsumierte Business-Events die an der Komponentenschnittstelle *angebotene* Funktionalität widerspiegeln, muss für

semantische Nachrichtentypen eines konsumierten Business-Events der operationale Subtypbegriff zu Grunde gelegt werden, sodass deren Substitutionsfähigkeit auf einer Generalisierung der Parametertypen sowie einer Spezialisierung des Rückgabetyps basiert. Der syntaktische Business-Event-Typ verhält sich folglich kovariant zu seinen semantischen Nachrichtentypen.

Im Gegensatz dazu reflektieren emittierte Business-Events die an der Komponentenschnittstelle *benötigte* Funktionalität, wobei sich die Substitutionsfähigkeit semantischer Nachrichtentypen eines emittierten Business-Events auf eine Spezialisierung der Parametertypen sowie einer Generalisierung des Rückgabetyps stützt. Dies entspricht dem Begriff des operationalen Supertyps, sodass sich der syntaktische Business-Event-Typ kontravariant zu seinen semantischen Nachrichtentypen verhält.

4.5.2.3 Definition Typart-2 (Verhaltenstyp)

Zwei syntaktische Business-Event-Typen i und j stehen in einer Typart-2-Beziehung, wenn die Protokollmengen von i und j in keiner Beziehung zueinander stehen, aber gilt:

$$Messages_i = Messages_j, \text{ wobei}$$

Typgleichheit für alle einander entsprechenden semantischen Nachrichtentypen gegeben sein muss.

Der formale Teil der Definition stellt auch hier wiederum sicher, dass die beiden syntaktischen Business-Event-Typen auf den gleichen semantischen Nachrichtentypen basieren.

4.5.2.4 Definition Typart-2 von Typart-1

Ein syntaktischer Business-Event-Typ j stellt eine Typart-2 einer Typart-1 eines syntaktischen Business-Event-Typs i dar, wenn die Protokollmengen von i und j in keiner Beziehung zueinander stehen, aber gilt:

$$Messages_i \subseteq Messages_j, \text{ wobei}$$

a) im Fall eines ***konsumierten*** *Business-Events für jeden semantischen Nachrichtentyp von i eine Subtyp-Entsprechung bei j existieren muss (Kovarianz).*

b) im Fall eines ***emittierten*** *Business-Events für jeden semantischen Nachrichtentyp von i eine Supertyp-Entsprechung bei j existieren muss (Kontravarianz).*

Analog zur Definition der Typart-1 sind auch hier mindestens die semantischen Nachrichtentypen des Basistyps vorhanden, wobei abermals eine Fallunterscheidung zwischen konsumierten und emittierten Business-Events vorgenommen werden muss.

4.5.2.5 Definition Kompatibilität

Zwei syntaktische Business-Event-Typen i und j sind kompatibel, wenn sie mindestens ein gleiches, gemeinsames Protokoll spezifizieren und alle in diesem Protokoll einander entsprechenden semantischen Nachrichtentypen (Zustandsübergänge) kompatibel sind.

Eine Kompatibilitätsaussage ist an dieser Stelle lediglich möglich, da durch die semantische Ebene festgelegt wird, ob ein Business-Event emittiert beziehungsweise konsumiert wird. Im Folgenden können nun ebenso Aussagen über den syntaktischen Typ abgeleitet werden.

4.5.3 Syntaktische Ebene

Auf der syntaktischen Ebene lassen sich folgende Vergleichsaussagen für den syntaktischen Nachrichtentyp ableiten:

Operationsmenge	Protokollmenge	KT∧KMT	synt. Nachrichtentyp
equal	equal	equal	equal
equal	equal	-	typekind-3
equal/extension	equal/extension	equal	typekind-1
equal/extension	equal/extension	-	typekind-3of-1
equal	-	equal	typekind-2
equal	-	-	typekind-3of-2
equal/extension	-	equal	typekind-2of-1
equal/extension	-	-	typekind-3of-2of-1

Tabelle 4.7
Vergleichsaussagen für den syntaktischen Nachrichtentyp

Tabelle 4.7 bezeichnet mit »KT« und »KMT« den Kommunikationstyp beziehungsweise den Kommunikationsmodelltyp, wobei die Bedingung »KT∧KMT = equal« die jeweilige Gleichheit für beide Attributwerte ausdrückt. Die nachfolgenden Definitionen basieren zunächst auf dieser Bedingung. Die Elemente der Operationsmengen (Operations) sind dabei Operationsty-

pen, für die jeweils typgleiche Elemente oder Supertyp- / Subtyp-Entsprechungen gesucht werden.

4.5.3.1 Definition Typgleichheit

Zwei syntaktische Nachrichtentypen i und j sind typgleich, wenn sie die gleiche Protokollmenge spezifizieren und gilt:

$$Operations_i = Operations_j, \text{ wobei}$$

Typgleichheit für alle einander entsprechenden Operationstypen gegeben sein muss.

Der formale Teil der Definition fordert an dieser Stelle, dass sich die beiden syntaktischen Nachrichtentypen aus den gleichen Operationstypen zusammensetzen.

4.5.3.2 Definition Typart-1 (Subtyp)

Ein syntaktischer Nachrichtentyp j stellt eine Typart-1 eines syntaktischen Nachrichtentyps i dar, wenn j eine gleiche oder erweiterte Protokollmenge von i spezifiziert und gilt:

$$Operations_i \subseteq Operations_j, \text{ wobei}$$

a) im Fall eines ***konsumierten*** *Business-Events für jeden Operationstyp von i eine Subtyp-Entsprechung bei j existieren muss (Kovarianz).*

b) im Fall eines ***emittierten*** *Business-Events für jeden Operationstyp von i eine Supertyp-Entsprechung bei j existieren muss (Kontravarianz).*

Die Typart-1 beinhaltet demnach zumindest die Operationstypen des Basistyps, sodass auch in diesem Kontext für den Fall einer Erweiterung der Operationsmenge der Subtyp wiederum »mehr« als der entsprechende Supertyp umfasst und damit eine erweiterte Typausprägung darstellt. Allerdings muss für die Elemente der Operationsmengen abermals eine Fallunterscheidung zwischen konsumierten und emittierten Business-Events vorgenommen werden, wobei sich dann neben dem Begriff der Erweiterung wiederum die Spezialisierung beziehungsweise Generalisierung als Subtyp-Charakteristik zeigt.

Für die Operationstypen eines konsumierten Business-Events wird – analog zur Nachrichtenebene – der operationa-

le Subtypbegriff zu Grunde gelegt, sodass sich deren Substitutionsfähigkeit auf eine Generalisierung der Parametertypen sowie eine Spezialisierung des Rückgabetyps stützt. Der syntaktische Nachrichtentyp verhält sich folglich kovariant zu seinen Operationstypen.

Die Substitutionseigenschaft von Operationstypen eines emittierten Business-Events basiert im Gegensatz dazu auf einer Spezialisierung der Parametertypen sowie einer Generalisierung des Rückgabetyps. Dies entspricht wiederum dem Begriff des operationalen Supertyps, sodass sich der syntaktische Nachrichtentyp kontravariant zu seinen Operationstypen verhält.

4.5.3.3 Definition Typart-2 (Verhaltenstyp)

Zwei syntaktische Nachrichtentypen i und j stehen in einer Typart-2-Beziehung, wenn die Protokollmengen von i und j in keiner Beziehung zueinander stehen, aber gilt:

$$Operations_i = Operations_j, \text{ wobei}$$

Typgleichheit für alle einander entsprechenden Operationstypen gegeben sein muss.

Der formale Teil der Definition stellt auch hier sicher, dass die beiden syntaktischen Nachrichtentypen auf den gleichen Operationstypen basieren.

4.5.3.4 Definition Typart-2 von Typart-1

Ein syntaktischer Nachrichtentyp j stellt eine Typart-2 einer Typart-1 eines syntaktischen Nachrichtentyps i dar, wenn die Protokollmengen von i und j in keiner Beziehung zueinander stehen, aber gilt:

$$Operations_i \subseteq Operations_j, \text{ wobei}$$

a) im Fall eines ***konsumierten*** *Business-Events für jeden Operationstyp von i eine Subtyp-Entsprechung bei j existieren muss (Kovarianz).*

b) im Fall eines ***emittierten*** *Business-Events für jeden Operationstyp von i eine Supertyp-Entsprechung bei j existieren muss (Kontravarianz).*

Analog zur Definition der Typart-1 sind auch an dieser Stelle mindestens die Operationstypen des Basistyps vorhanden, wobei

abermals eine Fallunterscheidung zwischen konsumierten und emittierten Business-Events vorgenommen wird.

4.5.3.5 Definition Typart-3 (technischer Typ)

Zwei syntaktische Nachrichtentypen i und j stehen in einer Typart-3-Beziehung, wenn sich diese bezüglich ihres Kommunikationstyps und / oder ihres Kommunikationsmodelltyps unterscheiden.

Die Typart-3 ist schließlich mit allen auf der syntaktischen Ebene definierten Typarten kombinierbar, wobei sich die auf Seite 199 in Abbildung 4.3 dargestellten Kombinationsmöglichkeiten ergeben.

4.5.3.6 Definition Kompatibilität

Zwei syntaktische Nachrichtentypen i und j sind kompatibel, wenn sie mindestens ein gleiches, gemeinsames Protokoll spezifizieren und alle in diesem Protokoll einander entsprechenden Operationstypen (Zustandsübergänge) kompatibel sind.

Eine Kompatibilitätsaussage auf dieser Ebene ist wiederum lediglich deshalb möglich, da bereits durch die semantische Ebene festgelegt wird, ob ein Business-Event emittiert beziehungsweise konsumiert wird. Bei unterschiedlichen Kommunikationstypen und / oder Kommunikationsmodelltypen ist die Kompatibilitätsbeziehung durch den implizit festgelegten Standardadapter gegeben (s.u.).

4.6 Konformität vs. Kompatibilität

Für ein Typmanagement ergeben sich an dieser Stelle die zentralen Fragestellungen, ob die Konformität eines Komponententyps zu einem anderen sichergestellt beziehungsweise die Kompatibilität zweier Komponententypen gewährleistet ist. Während die erste Fragestellung potenzielle *Substitutionskandidaten* in den Vordergrund stellt – also konforme Komponententypen, so fokussiert letztere potenzielle *Interaktionspartner* – in diesem Fall kompatible Komponententypen. Dabei bleibt festzuhalten, dass die Konformitätsbeziehung eine nicht-kommutative Rela-

tion darstellt[33], wohingegen die Kompatibilitätsbeziehung stets als kommutativ eingestuft wird.

Auf der semantischen Ebene ist die Unterscheidung dieser beiden Anfragen an einen Typmanager durch die Mengen der emittierten sowie konsumierten Business-Events (Domänenaktivitäten) gegeben. Substitutionskandidaten zeichnen sich dabei durch ein Interaktionsmuster aus, das mindestens dem Interaktionspotential sowie den Constraints des Basistyps Rechnung trägt, während Interaktionspartner hinsichtlich eines gleichen, gemeinsamen Protokolls auf einem inversen Interaktionsmuster basieren – in dem Sinne, dass emittierte Business-Events vom Interaktionspartner konsumiert werden können. Diese Bedingungen werden durch die Definitionen in Abschnitt 4.5.1 festgelegt. Auf der Nachrichtenebene wie auch auf der syntaktischen Ebene sind die entsprechenden Bedingungen durch die Definitionen in den Abschnitten 4.5.2 sowie 4.5.3 abgedeckt, die die Substitutionseigenschaft sowie die typsichere Interaktion auf technischer Ebene garantieren.

An dieser Stelle können schließlich allgemeine Aussagen über die Substitutionsfähigkeit der definierten Typausprägungen sowie über deren Interaktionsfähigkeit getroffen werden. Diese Aussagen können für jedes Wurzel-Element des Typbaums abgeleitet werden, sodass Schlussfolgerungen auf Basis einer Aussage

> »*A* ist kompatibel zu *B* und *A* hat eine konforme Typausprägung *C*. Folglich ist auch *C* kompatibel zu *B*.«

auf allen drei Ebenen des Typbaums gezogen werden können. Der in Abschnitt 6.2 vorgestellte Typmanager berechnet zum Registrierungszeitpunkt allerdings (Typvergleichs- und Kompatibilitäts-) Assoziationen zwischen allen registrierten Komponententypen einer Domäne, sodass bei der Konzeption dieser Typmanagement-Komponente keine derartige Schlussfolgerungsregel zu Grunde gelegt werden muss.

4.6.1 Substitutionskandidaten

Als Substitutionskandidaten, die nachfolgend eine typsichere Interaktion sicherstellen, kommen prinzipiell die folgenden *konformen* Typausprägungen in Frage, die gegebenenfalls automatisch adaptiert werden können.

[33] Im Spezialfall der Typgleichheit ist die Konformitätsbeziehung als kommutativ einzuordnen.

- typgleiche Ausprägungen
- Typart-1
- Typart-3 beziehungsweise Basistyp-3

Konforme Typausprägungen zeichnen sich durch Verhaltensäquivalenz zu ihrem Basistyp aus. Dies wird durch die Bereitstellung entsprechender Protokolle auf Seiten des Substitutionskandidaten realisiert, sodass diesem mindestens die Protokolle des Basistyps zu Grunde liegen. Für typgleiche sowie Ausprägungen der Typart-1 muss dabei gegebenenfalls die Adaption von Operationsnamen berücksichtigt werden, die automatisch vom Typmanager durchgeführt werden kann. Im Gegensatz dazu benötigen Ausprägungen der Typart-3 beziehungsweise des Basistyp-3 stets einen Standardadapter, der ihre Integration realisiert. Die Typart-2 stellt letztendlich keine konforme Typausprägung dar, weil sie keine Verhaltensäquivalenz garantiert.

In Analogie zur Aussage über konforme Typausprägungen kann nun ebenso eine Kompatibilitätsaussage abgeleitet werden.

4.6.2 Interaktionspartner

Eine allgemeine Aussage über potenzielle Interaktionspartner gestaltet sich an diesem Punkt der Diskussion weniger komplex als die entsprechende Aussage auf Seiten der Substitutionskandidaten, da keine Verhaltensäquivalenz sichergestellt werden muss. Für die Erfüllung der Kompatibilität ist auf den einzelnen Ebenen des Typbaums jeweils mindestens ein gemeinsames Protokoll vorausgesetzt.

Auf der semantischen Ebene muss dabei zusätzlich für das zu Grunde liegende Protokoll gelten, dass alle emittierten Business-Events vom Interaktionspartner konsumiert sowie alle konsumierbaren Business-Events auch ebenso emittiert werden, um die (abstrakte) Schnittstelle (Segment) vollständig zu nutzen (vgl. Abschnitt 4.5.1). Hinsichtlich des relevanten Protokolls kann somit von inversen Interaktionsmustern beider Interaktionspartner gesprochen werden.

Auf der Nachrichtenebene sowie der syntaktischen Ebene muss schließlich für alle semantischen Nachrichten- beziehungsweise Operationstypen des gemeinsamen Protokolls die Kompatibilität einander entsprechender Typausprägungen (Zustandsübergänge) erfüllt sein (vgl. Abschnitte 4.5.2 und 4.5.3). Nach der Behandlung dieser elementaren Fragen des Typmanagements

sollen im weiteren Verlauf Adapterkomponenten sowie die Realisierung von Polymorphismus thematisiert werden, wobei abschließend kurz auf den Begriff des Kompositionstyps und auf Typdomänen eingegangen werden soll. Letztendlich schließt das Kapitel mit der Ableitung diverser Systemanforderungen an eine Typmanagement-Komponente.

4.7 Adapter

Adapter wurden in Kapitel 3 als sinnvolle Interoperabilitätsmechanismen zur Erhöhung des Grades der Wiederverwendung bestehender Software dargestellt. An dieser Stelle sollen nun entsprechende Fragestellungen wieder aufgegriffen und das Adapterkonzept in ein Typmanagement integriert werden, das dann geeignet ist, zu entscheiden ob ein Adapter zwischen zwei – im Regelfall syntaktisch – inkompatible Komponenten geschaltet werden kann, um deren Inkompatibilität zu überbrücken[34]. Adapter werden im weiteren Verlauf der vorliegenden Arbeit nun ebenfalls als Komponenten aufgefasst, sodass sie – ebenso wie andere Komponenten – Typbeschreibungen als integralen Bestandteil besitzen, aus denen sich der *Adaptertyp* bestimmen lässt. Adapter*instanzen* können dabei ausschließlich für eine *konkrete* Technologie konzipiert werden, während sich der Adaptertyp durch Technologieunabhängigkeit auszeichnet. Wie die spätere Diskussion zeigen wird muss dem Adaptertyp dennoch ein Sonderstatus eingeräumt werden.[35]

Aufgrund ihres Sonderstatus werden Adaptertypen durch einen eigenen Namensraum[36] von anderen im Rahmen eines Typmanagements registrierten Komponententypen abgegrenzt. Dabei wird für jede Komponente zum Registrierungszeitpunkt

[34] Dies kann auf Grund des aggregierenden Charakters des Komponentenmodells (vgl. Kapitel 2) auch segmentweise im Sinne einer 1:n-Kopplung *mehrerer* Komponenten erfolgen. Daher ist zunächst die Beschränkung auf die Betrachtung von jeweils genau zwei Kopplungspartnern ausreichend.

[35] Insbesondere könnten Adapter eher auch *domänenübergreifend* zum Einsatz kommen (im Sinne wiederverwendbarer und häufig identifizierbarer Interaktions*muster* – vgl. a. die *Vergegenständlichung* von Entwurfsmustern in [Griffel, 1998]). Eine domänenübergreifende Verwendung »normaler« Komponenten ist dagegen auf Grund ihrer Ableitung aus der Aufbereitung jeweils eines spezifischen Anwendungsbereichs bestenfalls in seltenen Einzelfällen möglich.

[36] eine Typmanagement-*Domäne* im Sinne der Terminologie des Abschnitts 6.2

festgelegt, ob sie als Adapter oder als »normale« Komponenten (»Nicht«-Adapter) in die Wissensbasis eines Typmanagers aufgenommen wird. Somit muss diese Information nicht bereits in der Typbeschreibung repräsentiert werden. Die Kernidee dieser Unterscheidung ist dabei das weitere Zugriffs- bzw. Verwendungsrecht. Während ein Typmanager jederzeit den Zugriff auf die registrierten Komponenten erlaubt, um beispielsweise in einer Kompositionsphase deren Kompatibilität zu prüfen beziehungsweise Typvergleiche durchführen zu können, so verbietet er die gleichzeitige Einbeziehung der registrierten Adapter. Dadurch wird stets sichergestellt, dass eine Adapterkomponente nicht fälschlicherweise als ein potenzieller Interaktionspartner beziehungsweise Substitutionskandidat einer Komponente angesehen wird. In diesem Szenario hat deshalb ausschließlich ein Typmanager Zugriff auf die registrierten Adapter, da diese auf Grund ihres Sonderstatus einer speziellen Behandlung unterliegen müssen (s.a. Abschnitt 4.8.2).

Ferner sollten für jede Adapterkomponente auf der semantischen Ebene genau zwei (abstrakte) Schnittstellen (Segmente) gefordert werden, die von jeweils einer Komponente belegt werden. Durch diese Forderung ist ein Adapter genau dann ein möglicher Überbrückungsmechanismus, wenn beide Adapterschnittstellen von je einem Komponententyp *vollständig kompatibel* abgedeckt werden können und somit Kompatibilität für jedes syntaktische Element des Typmodells gegeben ist. Die Adapterkomponente kann dann ohne die Notwendigkeit eines *weiteren* Adaptionsmechanismus mit beiden Komponenten interagieren, sodass von einer weiteren Adaption einer Adapter-Komponenten-Kopplung – einer Adapterverkettung[37] – abgesehen werden kann.

4.7.1 Vollständige Kompatibilität

Zwei Komponententypen i und j sind genau dann vollständig kompatibel, wenn eine Kompatibilitätsbeziehung zwischen i und j auf der semantischen Ebene besteht sowie für jede dabei relevante

[37] In diesem Fall kann gegebenenfalls ein *kompositorischer* Adapter konstruiert werden, der wiederum (genau) zwei Segmente auf semantischer Ebene definiert. D.h. konzeptionell lässt der vorgestellte Ansatz solche Verkettungen zu. Da im Rahmen der vorliegenden Arbeit jedoch keine Untersuchung der Wirkung entsprechend langer Indirektionskaskaden auf die Performanz real implementierter Systeme vorgenommen wurde, soll hier pauschal die Forderung aufrecht erhalten werden, jeweils nur genau einen Adapter pro notwendiger Kopplung vorzusehen.

Domänenaktivität (Business-Event) Kompatibilitätsbeziehungen auf der Nachrichtenebene sowie auf der syntaktischen Ebene für alle einander zugeordneten Nachrichtentypen vorliegen. Auf der syntaktischen Ebene müssen zudem jeweils der Kommunikationstyp und der Kommunikationsmodelltyp übereinstimmen.

Alternativ wäre es ebenso denkbar, mehr als zwei Adapterschnittstellen zuzulassen. In diesem Fall würde allerdings die Komplexität der Prüfung einer (beidseitig-) kompatiblen Adapterkopplung stark ansteigen. Dazu müssten zusätzliche Metainformationen beziehungsweise Regeln definiert werden, die unter anderem festlegen, welche Adapterschnittstellen *obligatorisch* belegt sein müssen sowie *optional* belegt sein können.[38]

Im Gegensatz dazu weisen beliebig viele Adapterschnittstellen auf der Nachrichtenebene sowie der syntaktischen Ebene keine derartige Problematik auf, da bei deren Betrachtung bereits stets genau eine Schnittstelle auf der semantischen Ebene fixiert ist, sodass sich an dieser Stelle die Möglichkeit der Realisierung *generischer* Adapter ergibt. Im Hinblick auf die semantische Kompatibilität sind Adapter auf semantischer Ebene stets für einen speziellen Einsatz konzipiert, da die Adapterspezifikation die Semantikrepräsentation beider Interaktionspartner benötigt (vgl. Kapitel 3).

Letztendlich wird genau ein Adapter *vollständig* über alle Ebenen des Typbaums zwischen zwei Komponenten geschaltet, sodass beide Interaktionspartner entweder ohne oder sonst *ausschließlich* über die Adapterkomponente kommunizieren. Somit kann stets höchstens ein Adapter direkt zwischen zwei Komponenten geschaltet werden. Falls dabei bestimmte Interaktionen gegebenenfalls nicht adaptiert werden müssen, da hinsichtlich der betroffenen Teilfunktionalität keine Inkompatibilität besteht, hat der Adapter lediglich die Aufgabe der Weiterleitung des Informationsflusses der Interaktion (vgl. a. [Griffel, 1996]).

Ein Algorithmus zur automatischen Prüfung beidseitig-vollständiger Kompatibilität findet sich in Abschnitt 6.2.5.1.

[38] Dies wurde im Rahmen der in Kapitel 6 dargestellten Implementationsarbeiten auch in einem ersten Ansatz umgesetzt, auf eine vertiefte konzeptionelle Betrachtug wurde jedoch verzichtet, da diese Fragestellung nicht fundamental für das eigentlich im Mittelpunkt der Arbeit stehende generative Gesamtkonzept einer Softwarekonstruktion ist.

4.8 Mono- vs. Polymorphismus: Komponentenrollen

Analog zur Typisierung in Programmiersprachen kann eine Komponente *monomorph* – somit ist ihr genau ein Typ zugeordnet – wie auch *polymorph* sein. Bei Letzterem handelt es sich dann um die Zuordnung mehrerer Typen (Vielgestaltigkeit). Ein Polymorphismus bietet insbesondere die Möglichkeit einer Flexibilitätserhöhung hinsichtlich der Benutzung der entsprechenden Komponente, die – als polymorphe Komponente – verschiedene *Rollen* einnehmen kann.

Dabei werden nach [Cardelli & Wegner, 1985] verschiedene Varianten von Polymorphismus unterschieden.

- parametrischer Polymorphismus: Einführung von Typparametern zur Festlegung des Typs
- Inklusion: Subtypisierung, Vererbung
- Overloading: Neudefinition von (Teil-) Funktionalität (Anwendungskontext wird syntaktisch festgelegt)
- Coercion: Typumwandlung (Typkonversion)

Auf den parametrischen Polymorphismus soll an dieser Stelle nicht weiter eingegangen werden, da die vorliegende Konzeption des Typmodells keine explizite Definition von Typparametern vorsieht. Im Gegensatz dazu sind Inklusion und Overloading bereits durch das festgelegte Typmodell realisierbar. Die Komponentensicht verzichtet allerdings in der Regel auf Vererbungsbeziehungen zwischen Softwarebausteinen aufgrund der in Abschnitt 4.4.1 dargestellten Problematik. Somit müssen Typbeschreibungen redefiniert werden, um diese nachfolgend zu erweitern. Auf dieser Basis kann eine Subtypisierung etabliert werden, sodass eine Typausprägung einerseits im Subtyp-Kontext, andererseits aber auch im Kontext des Supertyps verwendet werden kann (Subtyp-Polymorphismus). Das Overloading kann schließlich in dem Sinne interpretiert werden, dass zwei Typausprägungen auf gleichen semantischen Repräsentationen mit *verschiedenen* syntaktischen Realisierungen basieren. Polymorphismus entsteht in diesem Fall durch die syntaktische Neudefinition eines Anwendungskontexts.

Des Weiteren ergibt sich der Anwendungsfall der expliziten Typumwandlung (Coercion), für den die zu Grunde gelegte Konzeption des Typmodells erweitert werden muss. Der

Polymorphismus-Begriff kann diesbezüglich in dem Sinne aufgefasst werden, dass sich die semantische Repräsentation (Ebene) einer Komponente austauschen lässt, wobei die Nachrichtenebene wie auch die syntaktische Ebene davon unberührt bleiben. Die Komponente nimmt in diesem Fall eine andere *Rolle* ein. Potenzielle Komponentenrollen werden dann in einer Menge zusammengefasst. Auf diese Weise unterstützt das Typmodell eine *Subjectivity* [Harrison & Ossher, 1993][Batory & Geraci,], d.h. die problem-, situations-, bzw. einsatzbezogene Schnittstellenauswahl einer Komponente (vgl. a. Abschnitt 5.7.1.1). Allerdings sollte für Komponenten, die eine derartige Typumwandlung zulassen, ein *Default-Typ* festgelegt werden, der die häufigste, »standardmässige« Verwendung kennzeichnet.

Komponenten, die keine Typumwandlung zulassen, zeichnen sich somit durch eine Nachrichtenebene aus, der genau eine semantische Ebene zugeordnet ist. Folglich ist die Business-Event-Syntax bezüglich ihrer Semantik *eindeutig*, während Komponenten, die eine Typkonversion gestatten, mehrere semantische Repräsentationen einer einzelnen Nachrichtenebene zuordnen. Diese Zuordnung wird bereits zum Modellierungszeitpunkt vorgenommen, sodass die Business-Event-Syntax im Hinblick auf ihre Semantik *mehrdeutig* ist. Die als Default-Typ ausgezeichnete Semantikrepräsentation referenziert dabei alle weiteren Komponentenrollen, die wiederum auf ihren Default-Typ verweisen. Nach der Typregistrierung im Rahmen eines Typmanagements ist allerdings lediglich der Default-Typ der Komponente im entsprechenden Namensraum (Typdomäne) sicht- und zugreifbar. Der Zugriff auf die der Komponente zusätzlich zugewiesenen Rollen ist dann über den Default-Typ möglich.[39]

Eine eindeutige Syntax-Semantik-Zuordnung impliziert an dieser Stelle nicht notwendigerweise den Monomorphismus der korrespondierenden Komponente, denn polymorphe Komponenten können auch aus den anderen Polymorphismus-Varianten hervorgehen. Anzumerken bleibt noch, dass instanzbezogene Informationen – wie beispielsweise der *aktuelle* Komponententyp (Rolle) – nicht innerhalb des Typmodells festgehalten werden.

4.8.1 Konsequenzen für ein Typmanagement

Durch die Möglichkeit der Modellierung typkonvertierbarer Komponenten ergeben sich diverse Auswirkungen auf ein Typmanagement. Die folgende Betrachtung setzt dabei voraus, dass

[39] vgl. a. Konzepte wie die Äquivalenzschnittstelle in CORBA-3.0 beziehungsweise das Query-Interface des COM+-Modells

alle Typen beziehungsweise Rollen einer Komponente verschieden voneinander sind.

Es stellt sich insbesondere die Frage, für welche Funktionalität einer Typmanagement-Komponente die Existenz weiterer Rollen typkonvertierbarer Komponenten relevant ist. Für Typvergleiche soll allerdings gefordert werden, dass diese ausschließlich auf dem Default-Typ der Komponenten operieren – ansonsten würde der Vergleich auf *Typmengen* basieren. Im Gegensatz dazu müssen Kompatibilitätsanfragen gegebenenfalls weitere Komponentenrollen berücksichtigen, falls eine derartige Beziehung nicht zwischen den Default-Typen zweier Komponenten evaluiert werden kann. Kompatibilität kann somit auch zwischen den Rollen zweier Komponenten zum Tragen kommen, wobei diese dann den entsprechenden *Anwendungskontext* bzw. die *Situation* einer Komponente ausdrücken. Der Kompatibilitätsbegriff lässt sich demnach dahingehend modifizieren, dass es *mindestens* einen Anwendungskontext gibt, in dem die Komponenten kompatibel zueinander sind.

Diese Konsequenz für den Kompatibilitätsbegriff betrifft schließlich auch die Suche nach potenziellen Interaktionspartnern zu einer selektierten Komponente sowie die Suche nach (beidseitig-) vollständig kompatiblen Adaptern für zwei zu koppelnde, inkompatible Komponenten. In beiden Fällen muss der Anwender eines Typmanagements zunächst den Anwendungskontext, in dem die Komponentenkopplung stattfinden soll, festlegen (Default-Typ vs. Rolle). Erst auf dieser Basis hat ein Typmanager die Möglichkeit, potenzielle Interaktionspartner beziehungsweise Adapter zu suchen, die genau in diesem Anwendungskontext verwendet werden können.

In diesem Zusammenhang müssen nun auch die in Abschnitt 4.7 betrachteten Adapterkomponenten diskutiert werden.

4.8.2 Adapter

Adapterkomponenten haben in der vorliegenden Konzeption einen Sonderstatus, der darin besteht, eine *Adaptionsfunktionalität* zur Verfügung zu stellen, während sich »normale« Komponenten (»Nicht«-Adapter) demgegenüber durch eine *Anwendungsfunktionalität* auszeichnen.

Diese Differenzierung impliziert insbesondere, dass Adapter beziehungsweise »normale« Komponenten in keinem Fall die jeweils andere Rolle einnehmen dürfen, denn dann wäre der zum Registrierungszeitpunkt im Rahmen eines Typmanagements vergebene Status nicht mehr sicherzustellen. Eine Kom-

ponente bietet somit *entweder* eine Adaptions- *oder* eine Anwendungsfunktionalität.

4.9 Kompositionstyp

Die Abgrenzung von Teilmengen über Domänenaktivitäten, Nachrichten sowie Operationen durch die entsprechenden Protokolle, die die Benutzung der (abstrakten) Schnittstellen (Segmente) auf den drei Ebenen des Typmodells spezifizieren, resultiert an dieser Stelle in der Möglichkeit, den *Kompositionstyp* zu konstruieren. Dieser kann schließlich aus einer Vereinigung der Teilmengen der Aktivitäten, Nachrichten beziehungsweise Operationen abgeleitet werden. Auf diese Weise können beliebige Segmente auf der jeweiligen Ebene des Typbaums kombiniert sowie neu zusammengefasst werden, sodass die daraus resultierende Protokollmenge eine neue, *kompositorische* Typausprägung kennzeichnet.

In der vorliegenden Arbeit soll ein solcher Kompositionstyp allerdings nicht weiter thematisiert werden, da diesem bei der grundsätzlichen Konzeption einer Typmanagement-Komponente einerseits keine fundamentale Bedeutung zukommt und er andererseits innere Abhängigkeiten zwischen Segmenten einer Komponente widerspiegeln würde (vgl. die Diskussion atomarer vs. aggregierter Komponenten in Kapitel 2).

4.10 Typdomänen

Typdomänen repräsentieren Namensräume, die es erlauben, Komponententypen zu Mengen zusammenzufassen und von anderen Namensräumen abzugrenzen. Typvergleiche und Kompatibilitätsprüfungen sowie die damit einhergehende Ableitung von Assoziationen werden dann ausschließlich innerhalb einer solchen Typdomäne durchgeführt. Typdomänen stellen die modellierten fachlichen »Welten« dar, die vom Nutzer eines Typmanagements anzulegen und bei jeder Typregistrierung explizit zu zuordnen sind. Sie lassen sich dabei hierarchisch strukturieren.

Auf dieser Basis ergibt sich wie oben angesprochen insbesondere für Adapterkomponenten die Möglichkeit ihrer Abgrenzung durch eine spezielle Typdomäne – die *Adapterdomäne*.

4.11 Bewertung

Die Auswahl von (vorgefertigten) Komponenten wird heute allgemein als kritischer, wichtiger werdender Punkt gesehen [Oberndorf et al., 1997], sowohl im Rahmen der Unterstützung von Entwicklungsumgebungen bzw. Software-Informationssystemen (vgl. Abschnitt 2.2) als auch entsprechenden »Komponentenmärkten«. Ganz im Sinne des Komponentengedankens ist das vorgestellte Typmanagement als explizite, eigenständige Komponente vielfältig einsetzbar, insbesondere auch auf Grund seiner flexiblen Abbildungsmöglichkeiten auf unterschiedlichste konzeptuelle und technische Klassifikationsarten. Das Typmodell kann dazu beitragen, bessere Tests zu erarbeiten und den Kontext einer Komponente besser abzugrenzen. Es leistet somit einen generellen Beitrag zu diversen Versprechungen des Komponentenansatzes.

Die Notwendigkeit der Typisierung von Komponenten reicht von ihrer Klassifikation zum Zweck einer Katalogisierung (etwa Produktkataloge), über die automatisierte Identifikation von Geschäftsprozessen (Workflows)[40] bis hin zu Anwendungskontexten wie etwa der Dienstvermittlung – dem sogenannten Trading – in offenen verteilten Systemen. Bei Letzterem handelt es sich um die Vermittlung eines entsprechenden Dienst*typs*, sodass in diesem Zusammenhang nicht der Name des Dienstes, sondern vielmehr dessen Typ herangezogen wird. Demnach kann auch von einem Typmanager-unterstützten Trading gesprochen werden [Müller–Jones et al., 1995].

Solche »Suchtechniken« wie das Trading und entsprechend konkrete Ansätze wie etwa Agora [Seacord et al., 1998], gehen jedoch nicht von einem übergreifenden Typbegriff sondern zunächst nur von der Auswertung einer Menge von Einzelattributen aus. Solche Attribute können dann zwar Methoden- oder Propertynamen sein, ein semantischer Zusammenhang und das Einbeziehen eines entsprechenden Gesamtverhaltens fehlen dann jedoch, da ein rein technisch orientiertes Typverständnis zu Grunde gelegt wird. Das hier vorgestellte Typmodell erlaubt dagegen auch explizit die Suche nach *verhaltensgleichen* Komponenten, deren Attribute sich dann unter Umständen gerade unterscheiden.

[40] Durch eine Typisierung der in einem Unternehmen zu findenden Einzelaktivitäten könnten diese durch nachfolgende Prüfungen auf Kompatibilität zu funktionalen Ketten im Sinne eines Workflows zusammengesetzt werden. Ein Test auf Konformität würde dabei (ersatzweise) alternative Beteiligte identifizieren.

Ein explizites Typmanagement, d.h. eines, das nicht mehr reiner Bestandteil einer konkreten Implementationstechnik (bzw. über eine Programmiersprache definiert) ist, erlaubt dabei zugleich einen flexibleren Umgang mit »klassischen« Typproblemen wie den Varianzfragen oder Sub- / Supertypverhältnissen, insbesondere durch den Rückgriff auf logische Contraints, die im Zweifelsfall erst eine Entscheidbarkeit für Konformitätsfragen ermöglichen.

Die entsprechenden Constraints werden dabei aus Sicht des Typmodells wiederum nur rein strukturell dargestellt, sodass dem Typmanagement bei Bedarf unterschiedliche konkrete Formalismen zur Verfügung gestellt werden können. Die in Abschnitt 3.10 eingeführten Policies nehmen dabei allerdings eine Sonderstellung ein, da sie zwar als logische Ausdrücke im Sinne einzuhaltender Richtlinien mit anderen denkbaren Constraints vergleichbar sind, allerdings der Instanzebene zu zuordnen sind, sodass sie zwar als Konfigurationsinformationen der Komponenten aufzufassen, aber als typirrelevant einzustufen sind. Ein Typvergleich *könnte* zwar zum Beispiel eine mögliche Unifikation zweier Policies im Sinne einer »Coercion« zum Erreichen einer Typkompatibilität oder auch -konformität heranziehen. Da die der Richtlinienverarbeitung unterworfenen Komponentenattribute jedoch jederzeit in der späteren Konstruktions- oder Laufphase einer Applikation noch der Modifikation durch den Konstrukteur oder Anwender unterliegen könnten, soll ihre Einbeziehung in den Typbegriff hier explizit vermieden werden.[41] Umgekehrt gibt es allerdings auch die Möglichkeit, die zunächst instanzunabhängigen Constraints – gegebenenfalls durch eine konkrete Wertebelegung – im Rahmen des Policymanagements einzusetzen. Die Randbedingungen auf semantischer Ebene könnten dann in Anlehnung an den Begriff des Business Events als »Geschäftsregeln« (Business Rules) aufgefasst werden. Eine nähere Untersuchung entsprechender Implikationen und Möglichkeiten ist jedoch nicht Gegenstand der vorliegenden Darstellung.

Durch die Einbettung von Automaten und Regeln sowie eines Rollenkonzepts handelt es sich letztlich um eine Form der modellbasierten Verifikation [Gluch & Weinstock, 1998], wobei einer möglichen Zustandsexplosion durch Schnittstellenzerlegung (Segmentierung) und Hierarchiebildung begegnet wird. Der Be-

[41] Ein Typmanager dürfte eigentlich schon dann keine Substituierbarkeit mehr zulassen, wenn ein angegebener Default-Konfigurationswert einer Komponente sich unterscheidet. Daher werden Attribute getrennt und erst auf Instanzebene betrachtet.

griff der Semantik wird dabei über die strukturelle Repräsentation entsprechender operationaler Konzepte definiert und so auch konkret einer maschinellen Verarbeitung zugänglich. Die vorliegende Arbeit weist die Tragfähigkeit des hier konzipierten Typmodells durch die Implementation einer prototypischen Typmanagement-Komponente nach (vgl. Abschnitt 6.2). Damit werden erstmals viele seit langem im Rahmen operationaler Modelle nur theoretisch diskutierte Ansätze in die Entwicklungspraxis getragen und insbesondere auch der Interaktionsbegriff greifbar konkretisiert.

Zur Benennung der strukturellen Repräsentationen stellt sich insbesondere hinsichtlich einer obligatorisch durchzuführenden Analyse der zu Grunde liegenden Domäne die Frage nach einer einheitlichen *Domänenterminologie* (vgl. Abschnitt 5.3). Für die geforderte Standardisierung von aus der Domänenanalyse hervorgegangenen Merkmalen erhebt sich dann die Frage, ob eine standardisierte Menge vordefinierter Merkmale mit festgelegter Semantik nicht in zu vielen Freiheitsgraden im Hinblick auf mögliche Merkmalsstrukturen resultiert. Somit wäre eine Standardisierung gesamter Merkmalsstrukturen für eine Domäne zu bevorzugen. Diese Entscheidung sollte durch das vorgeschlagene Typmodell jedoch nicht erzwungen werden, um beide Alternativen in der Praxis anbieten zu können.

Der Begriff des Typmanagements ist dabei nicht mit dem häufig im Kontext verteilter Objektsysteme verwendeten gleichzusetzen. Letzterer umfasst lediglich die Kompensation (auf entfernten Netzknoten) fehlender Klassendefinitionen, d.h. sorgt in der Regel für den Zugang zu entfernt bereitstehenden statischen Eigenschaften und Funktionen einer verteilten Gruppe von Objekten. Das Typmanagement hier ist dagegen weniger ein solcher Systemdienst als mehr ein Klassifikationswerkzeug. Auch nicht vorhandene Komponenten – also zunächst imaginäre, noch nicht implementierte – können einer System- bzw. Entwicklungsumgebung »bereitgestellt« werden, indem ihr Typ registriert wird. Sollten sie dann tatsächlich benötigt bzw. ausgewählt werden, kann die Typbeschreibung im Sinne einer Schnittstellen- und Verhaltensspezifikation als Ausgangsbasis für die konkrete Implementation einer entsprechenden Komponente dienen. Auf diese Weise können – unter Umständen aufwändige – Implementationsarbeiten verzögert werden, bis die tatsächliche Einsatznotwendigkeit sicher feststeht.

5 Konstruktion

Die Ergebnisse der in den voranstehenden Kapiteln dargestellten Untersuchungen sollten zwar ganz im Sinne des Komponentenparadigmas bereits für sich einen vollständigen und abgeschlossenen Lösungsansatz für die jeweils adressierten Probleme bereitstellen, dienten jedoch nicht in erster Linie diesem Selbstzweck. Die Nutzung des Komponentenbegriffs und die Festlegung seiner Modelleigenschaften, die Adaption von Verhalten und Eigenschaften solcher Bausteine und deren problemnahe Klassifikation dienten vielmehr der Vorbereitung der in diesem Kapitel erörterten Konzeption eines generativen Konstruktionsprozesses für Softwareapplikationen als eigentliches Ziel der vorliegenden Arbeit.

5.1 Motivation und Grundkonzeption

Die etablierte Herangehensweise aus anderen Industriezweigen sollte ein Anhaltspunkt sein, wie die Fertigung von Softwareanwendungen verbessert werden kann. Die Grundidee, dass unterschiedliche *Ausprägungen* einer Software, die sich jeweils nur geringfügig unterscheiden, prinzipiell aber stets das gleiche Problem adressieren, sich zu einer Entwicklungslinie zusammenfassen und gemeinsam analysieren lassen, verspricht insbesondere einen hohen Grad an Wiederverwendungspotential bestehender Bausteine. Aktuell beginnt sich daher in der Softwareentwicklung der Begriff der *Produktlinie* zu etablieren. Der Begriff verweist auf die gewünschte Ähnlichkeit zwischen der Produktion von Software mit der von anderen Industrieprodukten. Abschnitt 5.2.1 wird diese Aspekte vertiefen.

Eine Softwareanwendung ist in der Komponentensicht ein Kompositum aus mehreren Einzelbausteinen. Durch die Zusammenstellung einer *problemspezifischen Konfiguration* solcher – möglicherweise standardisierter – Einzelteile, kann dann eine, insbesondere auch kundenspezifische, *Individualsoftware* erstellt werden. Eine Analogie zu anderen Industrien möge dies

nochmals verdeutlichen: Ein Kunde hat beim Kauf eines Autos etwa die Wahl zwischen verschiedenen Motoren. Er kann sich für einen Diesel- oder einen Benzinmotor entscheiden. Je nach Wunsch des Kunden wird der entsprechende Motor in die Karosserie eingesetzt. Es muss allerdings gewährleistet sein, dass Motor und Karosserie auch zusammenpassen. Der Motor eines Formel 1-Wagens passt zum Beispiel nicht in die Karosserie eines Kleinwagens. Ausserdem kann der Kunde durch weitere Wünsche, die zum Beispiel die Farbgebung oder die Bauform (etwa Coupe oder Kombi) betreffen, sein individuelles Fahrzeug zusammenstellen, allerdings unter einer bewussten Einschränkung der Kombinationsmöglichkeiten beispielsweise durch (produktions-) technische Randbedingungen. Er kann jedoch auch optionales Zubehör wie Klimaanlage und beheizte Sitzkissen auswählen. Eine analoge Vorgehensweise sollte in der Produktion von Softwareapplikationen realisiert werden und ist Gegenstand der weiteren Betrachtungen.

Der Herstellungsprozess selbst sollte dabei weitgehend automatisiert werden. Ein Blick auf die Musikauswahl heutiger Radiosender verdeutlicht dass dann usätzliche Informationen unterschiedlichen Charakters bereitgestellt werden müssen, um automatisch sinnvoll erstellte Ergebnisse zu erhalten. Die meisten Radiosender stellen ihr Musikprogramm nicht mehr per Hand durch die Moderatoren zusammen. Vielmehr sind hierfür Computerprogramme zuständig, die die Musiklisten zusammenstellen. Damit die ausgewählte Musik zusammenpasst, ist es notwendig, dass dem Programm mehr Informationen über die Musiktitel zur Verfügung stehen als der Name des Interpreten und des Liedes. Die Länge des Stückes ist notwendig, damit die gesamte Musikliste die richtige Länge hat. Damit auch die Musikstile und die Atmosphäre der Stücke zusammenpassen, müssen dem Programm die entsprechenden Daten zur Verfügung stehen und sie müssen ausgewertet werden können. Je mehr Informationen über die einzelnen Musikstücke vorliegen, desto gezielter kann das Programm individuelle und auch zusammenpassende Musiklisten erstellen. Aus informationstechnischer Sicht liesse sich dieses Szenario als *deklarative* oder auch deskriptive Zusammenstellung bezeichnen (vgl. a. Abschnitt 6.2.1.2).

[Huber & Seidl, 1999] nennen die Geschichte der Informatik eine Geschichte der Suche nach Abstraktion von der eigentlichen Implementierung. Programmiersprachen haben eine Entwicklung von maschinennahen Sprachen zu objektorientierten Sprachen genommen und unterliegen derzeit der weiteren Abstraktion im Sinne des Komponentengedankens [Griffel, 1998].

Eine Spezifikation von Anwendungen lediglich durch Angabe von gewünschten *Merkmalen* der jeweiligen Problemlösung wie sie diese Arbeit betrachtet, abstrahiert nochmals erheblich weitergehend von programmiertechnischen Details. Auf diese Weise kann der Betrachtungsfokus der Entwicklung – die stets als Problemlösung und nicht als Selbstzweck verstanden werden sollte – enger auf die *fachlichen* (domänenspezifischen, s.a. Abschnitt 5.3) Erfordernisse eingestellt werden.

Die Erstellung von individuellen Softwareanwendungen einer Produktlinie erfolgt auf eine jeweils sehr ähnliche Weise. Es fällt daher leichter als in einer völlig allgemeinen Softwareentwicklung, Mechanismen zu definieren und zu realisieren, die eine weitgehend automatisierte Erstellung solcher Anwendungen nach vorgegebenen (Architektur-) Mustern ermöglichen. Dies resultiert in einem *generativen* Vorgehen.[1] Eine solche Vorgehensweise würde einen Generator einsetzen, der dann jedoch zusätzliche Informationen benötigt, etwa wie die einzelnen Komponenten zu verwenden sind und welche Eigenschaften sie kennzeichnen. Es reicht nicht aus, zu wissen, welche Programmierschnittstellen (APIs) die einzelnen Komponenten anbieten, sondern zusätzliche semantisch reichere Informationen sind notwendig, um so mehr, als dass die hier postulierte Komponentensicht einen »black-box«-Charakter fordert und somit dem Anwender nur die explizit gemachten Informationen zur Verfügung stehen.

Entsprechende Informationen sollten dabei nach Möglichkeit »nah« bei der konkreten Software gehalten – z.B. in Form erweiterter Schnittstellenbeschreibungen – nicht jedoch in unübersichtlichen Handbüchern versteckt werden. Die vorangegangen Kapitel haben entsprechende Vorabeiten geleistet. Eine solche enge Assoziation beschreibender Informationen wird insbesondere auch dem Anspruch eines besseren Übergangs zwischen Modell- und Implementationssicht gerecht, indem der deskriptive Anteil der Software einen direkten Einfluss[2] auf sie ausüben kann etwa durch Auswertung während der Installationsphase oder auch der späteren Laufzeit, wie es etwa moderne Applikationsserver-Architekturen wie EJB [Sun, 2000] oder Microsoft MTS [Kirtland, 1997] in Form einer attributbasierten Programmierung vorsehen).

Ein *Generator* benötigt dann lediglich ein »Grundgerüst« für eine spezifische Linie von Anwendungslösungen und kann dann

[1] Der Begriff *generativ* soll dabei in dieser Arbeit nicht die »Generationen« der Einzelprodukte einer Entwicklungslinie betonen, sondern auf den *erzeugenden* Charakter der Konstruktion hinweisen.

[2] vgl. a. Abschnitt 3.10

– eventuell unter dem Einfluss individueller Entwicklerangaben – Bausteine aussuchen und zu einer Anwendung zusammenstellen. Der Generator erstellt dabei nur ein Rahmenprogramm (Anwendungsgerüst), das die einzelnen Komponenten miteinander verknüpft und im Idealfall selbst völlig frei von Anwendungslogik ist (vgl. a. die Diskussion von Applikation*skeletten* in [Griffel, 1998]).

Eine Generierung von Anwendungen hat den Vorteil, schon während der Erstellung der Software Fehler durch menschliche Fehlentscheidungen ausschließen und vermeiden zu können. Im Gegensatz dazu steht heute immer noch die »Rettungsmassnahme danach«, also das Testen und Debuggen einer Software *nach* ihrer Fertigstellung im Vordergrund der traditionellen Softwareentwicklung.[3] Die Erstellung von Softwareanwendungen sollte von menschlichen (Fehl-) Entscheidungen in der technischen und formalen Ebene befreit werden. Der Programmierer sollte sich verstärkt auf die fachliche Ebene konzentrieren können. Der Einsatz von Generatoren verspricht eine definierbare Qualität der Software, da der Prozess ihrer Erstellung messbarer wird.

Ziel der vorliegenden Arbeit ist die Erarbeitung aller zur prototypischen Umsetzung eines vollständigen generativen Konstruktionsprozesses notwendigen Konzepte und Techniken, insbesondere eines Generators, der es erlaubt, durch eine *angeleitete* Auswahl von Eigenschaften der gewünschten Anwendung,[4] eben diese zu generieren. Die Anwendung soll dabei aus lose zu koppelnden, vorgefertigten Komponenten bestehen. Der Generator muss sicherstellen, dass die Komposition von Komponenten auch im Sinne einer *Verifikation* eine korrekte Anwendung ergibt, d.h. eine sinnvolle und eine im Sinne einer *Validierung* das gestellte Problem lösende Verarbeitung erlaubt. Eine Validierung kann allerdings nicht mehr allgemeingültig durch einen Generator vorgenommen werden. Ein entsprechendes Domänenwissen (s.a. Abschnitt 5.3), auf das der Generator ohnehin Zugriff

[3] Vgl. aber a. die aktuellen Bemühungen zur Etablierung *früher*, d.h. die eigentliche Entwicklung *begleitender* Tests etwa im Rahmen des eXtreme Programmings (s.a. Abschnitt 1.1.5). Solche »Unit-Tests« entsprechen allerdings dann genau der Komponentensicht mit ihren vorgetesteten Einzelbausteinen.

[4] Der interessante Aspekt hierbei liegt in der Zuordnung (Abbildung) von Anwendungsmerkmalen auf die Eigenschaften der konkreten Einzelbausteine. So kann ein Merkmal der Anwendung gewissermassen komponentenüberspannend zum Beispiel Einfluss auf verschiedene Eigenschaften zweier Komponenten haben.

hat, lässt allerdings die Generierung von Testfällen zumindest für Teilfunktionen denkbar werden.

Die Funktionalität der Komponenten selbst ist als getestet anzusehen, aber es muss sichergestellt werden, dass die ausgewählten Komponenten auch zueinander kompatibel sind. Die Komponenten benötigen für eine automatische Komposition eine ausführliche Beschreibung ihrer Funktionen und Eigenschaften, um so mehr als dass ein Generator in der Lage sein muss, diese Einzelteile mit einer Position in der ihm vorgegebenen Architektur einer Anwendungslinie zu assoziieren. Inspiriert ist diese Vorgehensweise von einer modellbasierten Verifikation [Gluch & Weinstock, 1998].

Im Gegensatz zu den anderen weiter unten noch vorgestellten generativen Ansätzen in der Literatur steht hier die Erstellung von Anwendungen und nicht von Komponenten im Vordergrund. Der Generator erstellt somit eine fertige Anwendung, die direkt verwendet werden kann. Es wird ausserdem Wert auf die Flexiblität des Generators gelegt, sodass er nicht nur für eine einzige Domäne zum Einsatz kommen kann. Stattdessen wird eine domänenspezifische Parametrisierung des Generators angestrebt, sodass das Resultat eher als *Meta-Generator* bezeichnet werden kann. Die vorliegende Arbeit setzt dabei eine vorangegangene Domänenanalyse (vgl. a. Abschnitt 5.3) voraus, die im Rahmen der weiteren Darstellung nur exemplarisch betrachtet wird. Insbesondere werden entsprechende Vorgehensmethoden und Modelle zur Erschliessung einer Domäne nur am Rande gestreift, um den Rahmen der Darstellung nicht zu sprengen. Allerdings werden verschiedene weitere Mechanismen angesprochen und eingeordnet, die eine entsprechend weitergehende Betrachtung unterstützen.

5.1.1 Überblick

Im Folgenden werden zunächst die Konzepte der Produktlinien- bzw. Anwendungsfamilien-orientierten Entwicklung sowie Aspekte des Begriffs der Architektur diskutiert. Die entsprechende Aufbereitung eines Anwendungsbereichs (das »Domain Engineering«) wird erörtert und sein Bedeutung für einen generativen Ansatz betont. Generative Ansätze selbst werden vorgestellt und vom eigenen Ansatz des Autors abgegrenzt sowie aufgezeigt, welchen einflussnehmenden Konzepten wie etwa der Aspektorientierung bzw. der adaptiven Programmierung er unterliegt. Es wird verdeutlich, dass alle modernen Konzepte dieser Art auf ähnlichen Ideen beruhen bzw. ähnliche Proble-

matiken identifizieren und vergleichbare oder sich ergänzende Lösungsansätze hervorgebracht haben. So haben generell moderne Ansätze etwa stets eine »Trennung der Belange«, d.h. der unterschiedlichen Aspekte einer Gesamtlösung vor Augen.

In Abschnitt 5.8 werden die Konzepte, die dem später implementierten Generator zu Grunde liegen, vorgestellt und die Erstellung einer individuellen Anwendung durch einen Softwarekonstrukteur mittels einer angeleiteten Spezifikation von gewünschten Eigenschaften dargestellt. Die Auswertung der dabei notwendigen Meta-Informationen aus dem vorgestellten Typmodell wird erläutert. Die notwendige Erweiterung entsprechender Komponenten-Deskriptoren wird vorgenommen.

Nach der Darstellung dieser Grundkonzepte eines Generators, wird zur Illustration des Gesamtprozesses eine exemplarische Domänenanalyse am Beispiel einer Taschenrechneranwendung durchgeführt. Es wird aufgezeigt wie auf Basis der erhaltenen Ergebnisse, ein Taschenrechner mit Hilfe des Generators implementiert werden kann. Die notwendigen Komponenten eines Taschenrechners werden zur Illustration ebenfalls exemplarisch beschrieben und eine entsprechende Anwendungsarchitektur entwickelt.[5]

5.2 Produkte und Architekturen eines Anwendungsbereichs

5.2.1 Produktlinien

Der Begriff der Konstruktion in Abgrenzung zur üblichen Programmierierung wird heute hilfreich ergänzt durch das in den letzten Jahren rasch an Bedeutung gewinnenden Konzepts der *Produktlinie* (engl. product line) in der Softwareentwicklung.[6] Hinter diesem Begriff steckt zwar keine eigentliche Innovation,[7]

[5] Das Beispiel eines Taschenrechners wurde hier gewählt, da es »klein« genug ist, um im Rahmen der Darstellung vollständig wiedergegeben werden zu können und so alle auftretenden Aspekte konkret diskutieren zu können. Das Gesamtkonzept der Arbeit wurde auch mehrfach für ausgesprochen komplexe Anwendungen eingesetzt (vgl. Kapitel 7).

[6] Eine allgemeine Diskussion zu Produktlinien und den Aspekten der Einführung einer Produktlinien-orientierten Entwicklung findet sich beispielsweise in [Bass et al., 1999] und [Cohen, 1999].

[7] So findet sich das Konzept eines »Family Design« etwa schon in [Parnas, 1976].

aber er fördert den methodischen Einsatz von Komponenten in der Praxis. [Clements & Northrop, 1997] definiert eine Software-Produktlinie als eine Serie von Systemen, die gemeinsame Eigenschaften, die spezielle Bedürfnisse eines bestimmten Marktsegments erfüllen, miteinander teilen (vgl. a. [Bass et al., 1997]). Es soll hierbei nicht jede Software einzeln neu entwickelt werden, sondern aus geeigneten Komponenten bestehen, die durch vorgesehene Variationsmöglichkeiten angepasst und in einer produktlinienweiten *Architektur* eingesetzt werden. Die Entwicklung neuer Software wird dadurch mehr zu einer Frage der Zusammenstellung als der Erschaffung – ganz im Sinne des Komponentenparadigmas. Die entscheidende Entwicklungstätigkeit ist eher in der Integration als einer Programmierung zu sehen. Es ist dann sehr wichtig, den Aufbau der Software auch so zu gestalten, dass er auf den Einsatz von Komponenten ausgerichtet ist.

In einer Produktlinie erfolgt die Entwicklung einer neuen Version einer Software durch die Weiterverwendung einzelner Bausteine, sowie der Architektur der Vorgängerversion. Der Umfang typischer, in der Praxis zu beobachtender Produktlinien liegt dabei in der Größenordnung von 1 bis 60 Applikationen, höhere Werte bilden eher die Ausnahme [Bass et al., 1997]. Es werden nur einzelne Bestandteile der Anwendung entfernt, durch neue Versionen ausgetauscht oder ergänzt, in der Regel in einer iterativen Vorgehensweise. Die Entwicklung sollte so ausgerichtet sein, dass möglichst viele Komponenten ähnlicher Anwendungen übernommen werden können. Es wird eine Architektur zu Grunde gelegt, in deren Rahmen Komponenten zu verschiedenen Produkten kombiniert werden. Eine solche Vorgehensweise kann dann zu Recht als »architekturzentriert« bezeichnet werden. Die gemeinsame Architektur erlaubt die ökonomische und weniger fehlerträchtige Ableitung zusätzlicher Anwendungen [Bass et al., 1997]. Auf diese Weise wird wesentliches Know-how der Entwicklung gesichert, das vor allem auch bei einem Technologiewechsel erhalten bleibt, da wesentliche Arbeit in die Erschliessung einer Domäne und die Definition dort gültiger und sinnvoller Architekturen geflossen ist. Diese Ergebnisse sind in der Regel deutlich langlebiger als die Details konkreter sich rascher wandelnder Technologien.

Für eine Produktlinie stellt sich dabei die Frage, ob extra entworfene oder aufgefundene (engl. *designed vs. mined* Komponenten verwendet werden. Erstere sind oft dem zu lösenden Problem angemessener, letztere oftmals besser getestet und damit von nachgewiesener Qualität bzw. erlauben kürzere Entwick-

lungsphasen.[8] Eine entsprechende Wieder- bzw. Weiterverwendung verbessert und bewahrt vor allem Erfahrungen der Entwicklung, da sich mehr Ergebnisse projektübergreifend ansammeln. Die Produktliniensicht fördert damit ein projektübergreifendes Arbeiten, bzw. führt auf natürliche Weise zu einer Komponentensicht, da übergreifend, wiederverwendbar und modular gedacht werden muss. Umgekehrt erlaubt die Komponentensicht eben gerade eine entsprechende Produktlinienentwicklung, sodass die zugehörigen Konzepte zunehmend im Rahmen des Komponentenparadigmas diskutiert werden.

5.2.2 Produktfamilien

Bei der Autoproduktion werden neue Autos auf Basis vorhandener Produkteile gebaut. Die Neuentwicklung eines Sportwagens setzt zum Beispiel auf dem Grundgerüst eines Mittelklassewagens auf. Ein Motor wird auch nicht für einen speziellen Autotyp entwickelt, sondern wird in mehreren Autos eines Herstellers eingesetzt. Es wird versucht, so weit wie möglich auf vorhandene Produktteile aufzusetzen, um einerseits Entwicklungszeiten und Produktionskosten zu senken und andererseits bestehende Erfahrungen und Qualitäten zu sichern. Die gleiche Praxis sollte nach Meinung des Autors wiederum auch bei der Entwicklung von Softwaresystemen Einzug finden. Die Softwareentwicklung sollte in diesem Sinne von der klassischen Produktherstellung lernen und einer Industrialiserung unterzogen werden.

Meist bezieht sich der Begriff der Produktlinie nur auf ein Produkt und seine Nachfolger, die auf dem gleichen Grundgerüst und einigen wiederverwendeten Komponenten beruhen. Entwicklungslinien sollten allerdings auch *artverwandte* Produkte berücksichtigen. Eine Menge von Systemen aus einer Domäne kann mit hoher Wahrscheinlichkeit von gemeinsamenen Bausteinen profitieren.[9] Es wird hierfür dann eine ganze *Produktfamilie* (engl. *Product Line Family*) betrachtet. Dabei wird versucht gemeinsame Komponenten zu identifizieren und zu entwickeln. Anwendungen von Produktfamilien sollen zu einem grossen Teil aus diesen gemeinsamen Bausteinen bestehen. Es gibt für jedes Einzelprodukt eine Architektur, die den Rahmen des sinnvollen und problembezogenen Einsatzes diese Komponenten vorgibt. Die Software selbst kann dann schrittweise durch Verbesse-

[8] und damit ein besseres »Time-to-Market«

[9] Domänenübergreifend nimmt der Wiederverwendungsgrad rapide und bis zur Bedeutungslosigkeit ab und wird daher im Rahmen dieser Arbeit nicht weiter betrachtet.

rung der Einzelkomponenten weiterentwickelt werden. Es wird insgesamt eine strategische Wiederverwendung angestrebt.

Zur besseren Abgrenzung der möglichen Ausprägungen von Produktlinienarchitekturen wird im weiteren Verlauf der Arbeit der Begriff Produktfamilie verwendet, wenn auch artverwandete Produkte bei der Konzeption der Architektur mit einbezogen werden. Es wird bislang bei Vorgehensmethoden für Produktlinien und Produktfamilien nicht näher spezifiziert, wie die zugehörigen Architekturen auszusehen haben oder wie sie erarbeitet werden sollten. Es gibt auch noch keine allgemein gültigen Hinweise wie Einzelbausteine mit den Architekturmodellen assoziiert werden sollten. Der Ansatz dieser Arbeit muss diese Fragen aber im Rahmen der Konzeption und Realisierung eines Softwaregenerators beantworten. Als Ausgangsbasis soll zunächst im Folgenden eine kurze Betrachtung des Standes der Architekturdiskussion erfolgen.

5.2.3 Architekturbeschreibungssprachen

Die zentrale Rolle der Architektur[10] für eine Produktlinie wurde bereits betont. Es sollte möglich sein, komplexe Systeme auf einem entsprechend hohen Abstraktionsniveau in ihren Strukturen mit Hilfe einer Architektur zu beschreiben, wobei sich jedoch die Frage stellt, wie die Architekturen selbst ausgedrückt werden können. Aus dem akademischen Umfeld der neunziger Jahre sind zahlreiche *Architekturbeschreibungssprachen* (engl. *Architecture Definition Languages* (ADL)) wie etwa Aesop [Garlan et al., 1994]), Meta-H [Vestal, 1996]), C2 [Medvidovic et al., 1997]), Rapide [Luckham et al., 1995], [Team, 1997]), SADL [Moriconi et al., 1995]), UniCon [Shaw et al., 1995]) und Wright [Allen & Garlan, 1994]) hervorgegangen.[11] Diese ADLs beschreiben alle die Systemstruktur

[10] [Shaw, 1990] definiert den Architekturbegiff als »software design at a level of abstraction that focuses on the patterns of systems organization that describes how functionality is partitioned and how those elements are interconnected.« [Peterson & Stanley, Jr., 1994] identifiziert demgemäss eine Aufteilungsstrategie und ein Koordinationmodell als die beiden Schlüsselfaktoren eines organisatorischen Musters wie es eine Architektur in diesem Sinne darstellt. Der Begriff soll in dieser Arbeit nicht näher diskutiert werden, aber entsprechend der vorstehenden Definition von allgemeineren Architekturbegriffen anderer (Wissenschafts-) Disziplinen abgegrenzt werden.

[11] Für eine allgemeine Betrachtung und Zusammenfassung s.a. [Clements, 1996].

als eine hierachische Komposition von interagierenden Komponenten, die durch Konnektoren miteinander verbunden werden. Neben dem Strukturskelett können zusätzliche semantische Informationen ausgedrückt werden. Diese hängen jedoch von der jeweils verwendeten Beschreibungssprache ab.

Eine typische ADL kommt in Verbindung mit einem konzeptionellen Framework, einer konkreten Syntax für die Charakterisierung von Systemarchitekturen, sowie Werkzeugen, die zum Beispiel die Analyse und die Simulation unterstützen, zum Einsatz. Die einzelnen angeführten ADLs haben unterschiedliche Schwerpunkte. Aesop etwa bietet die Möglichkeit, Familien von Architekturen zu definieren. Rapide bietet dafür Werkzeuge für eine Simulation der Anwendung an. Einige Tools bieten die Möglichkeit, eine Architektur zu analysieren und zum Beispiel auf Konsistenz zu untersuchen.

Es wurde mit ACME [Garlan et al., 1995b], [Garlan et al., 1997]) ein Austauschformat entwickelt, dass es erlaubt, die Daten zwischen den Werkzeugen der einzelnen ADLs auszutauschen. ACME bietet eine standarisiserte Form zur Beschreibung der Struktur, sowie ein offenes System zur Beschreibung von zusätzlichen Informationen. Dennoch spielen ADLs in der industriellen Entwicklungspraxis heute so gut wie keine Rolle. Dies liegt zum einen sicherlich daran, dass das »Architekturbewusstsein« bei den Entwicklern noch unzureichend (oder gar nicht) ausgeprägt ist – die Komponentensicht versucht dies ja gerade in den entsprecheden Köpfen zu verankern. Zum anderen geben die genannten ADL-Ansätze wenig Hinweise bzw. Unterstützung in der Verbindung zwischen Architekturgerüst und *konkreten Komponenten* zur Implementation. Meist wird der Begriff der Komponente nur sehr allgemein verwendet und der Schwerpunkt auf formale Konsistenzbetrachtungen der Beschreibungssprachen selbst gelegt.

Eine Architektur kann aber auch direkt durch die Angabe von Komponenten und deren Kommunikation untereinander festgelegt werden. Dafür existieren dann die Konstrukte »Komponente«, die die Basisfunktionalität aber auch zum Beispiel die Datenbanken eines Systems repräsentiert, und »Konnektor«, der Komponenten miteinander verbindet und die Verknüpfung, bzw. die mir ihr verbundenen *Interaktionsmuster* explizit repräsentiert. Bei der Interaktion kann es sich beispielsweise um simple Methodenaufrufe, Events und Pipelines, sowie komplexere Client-Server-Protokolle handeln (vgl. a. Kapitel 2). Auf diese Weise werden strukturelle und kontextabhängige Informationen, in Form der Konnektoren, von den funktionalen und kontextun-

abhängigen Informationen – der eigentlichen Komponentenfunktionalität – getrennt.

Der Begriff der Komponente beschreibt in ADLs eine bestimmte Funktionalität, die durch einzelne Bausteine erfüllt werden kann. Diese Bausteine können Komponenten im Sinne dieser Arbeit sein. Es kann sich jedoch auch zum Beispiel um Codezeilen oder komplette Anwendungen handeln. ADLs betonen nur die eigentliche Architektur. ADLs sind damit aber näher an einer komponentenbasierten Entwicklungssicht als es beispielsweise typische, auf einer objektorientierten Modellierung basierende Entwicklungsprozesse erlauben (s.a. Abschnitt 6.4.3). So wird etwa die *Unified Modeling Language* erst in der Version 2.0 weitere Konzepte für den Einsatz expliziter Komponenten berücksichtigen. Es wird sich zeigen, ob auch die zugehörigen Entwicklungsprozesse den Komponentengedanken und eine wirklich architekturzentrierte Sicht besser berücksichtigen werden.[12]

5.2.3.1 Bewertung und Abgrenzung

Praktische Implikationen und Anwendungsmöglichkeiten wurden bisher stets von der Forschung im ADL-Bereich vernachlässigt [Clements & Weiderman, 1998]. Der Ansatz der vorliegenden Arbeit verzichtet aus diesem Grund auf eine explizite ADL und überlässt es den Komponenten-Deskriptoren (aus der Entwicklungspraxis stammend, vgl. Abschnitt 5.9) die notwendigen Informationen über Verknüpfungen und Randbedigungen zu liefern.

Während ADLs als *zusätzliche* Beschreibung dienen, um den Architekturstil eines Systems, d.h. seine Zerlegung in Komponenten und Konnektoren, zu beschreiben und zu formalisieren, geht der hier vorgestellte Ansatz davon aus, dass viele Detailfragen, die durch eine ADL erfaßt werden, bereits durch die vollständige Beschreibung einer Komponente »mitgebracht« werden, sodass die Kombination mit anderen Komponenten lediglich noch einen Rahmen, nicht mehr aber die Spezifikation der Details der durchzuführenden Verknüpfungen berücksichtigen muss. Die Idee der Externalisierung und Explizität der Architektur eines System wird vom Autoren absolut befürwortet und nicht Frage gestellt, allerdings ist eine direkte Integration *in* die fertige Software notwendig. Daher umfasst das vorgestellte Typmodell etwa Interaktionsprotokolle (vgl. [Allen, 1997]), aber eben nicht als Teil einer ADL sondern als Teil der zu verarbeitenden

[12] Allgemeine und historische Betrachtungen zur einer Architekturzentrierten Entwicklung finden sich zum Beispiel in [Clements, 1996].

Komponenten. Am ähnlichsten ist der hier vertretene Ansatz damit noch Shaws UniCON (vgl. [Allen, 1997]). Aus der Diskussion der ADLs kann auf jeden Fall die Erkenntnis übernommen werden, das Architekturstile als Prädikate über Systemkonfigurationen betrachtet werden sollten (vgl. a. den Begriff des Architekturtyps in [Horn et al., 1998]). Die entsprechende Stildefinition für Architekturen durch eine Menge von Typen in Aesop [Allen, 1997] hat dabei den hier vorgestellten Ansatz inspiriert. Eine Architektur ist in diesem Sinne lediglich ein syntaktischer Rahmen, der das Einhängen von Semantiken (via Komponenten) erlaubt und so zu einem bedeutungstragenden System wird.

ADLs geben zunächst nur eine Antwort auf die Konsistenz einer solchen Architektur, lösen aber nicht das Problem geeignete Komponenten zu finden. Der in Kapitel 4 vorgestellte Klassifikationsansatz ist dagegen zum Beispiel geeignet, um ADL-basierten Werkzeugen auch die *Auswahl* von Komponenten zu erlauben.

5.3 Aufbereitung eines Anwendungsbereichs

Um überhaupt sinnvolle Systemarchitekturen eines Problembereichs identifizieren zu können, ist ebenfalls ein methodisches Vorgehen notwendig. Da der Ansatz für eine Generative Softwarekonstruktion in der vorliegenden Arbeit auf eine Aufarbeitung der jeweils fraglichen Anwendungsdomänen aufsetzt, andererseits nach Meinung des Autors diese Problematik entgegen ihrem Stellenwert noch viel zu selten in der Literatur vertreten ist, soll an dieser Stelle etwas ausführlicher auf ein entsprechendes *Domain Engineering* eingegangen und die zum Teil bereits verwendeten Begriffe und Konzepte verdeutlicht werden. Die Darstellung orientiert sich dabei an [Czarnecki & Eisenecker, 2000], die eine ausgezeichnete Zusammenfassung der Thematik bieten.

Unter dem Begriff *Domain Engineering* wird die Sammlung, Verwaltung und die systematische Wiederverwendung von Artefakten und Erfahrungen für die Entwicklung von Systemen oder Teilsystemen einer bestimmten Domäne verstanden. Artefakte sind beispielsweise Dokumente, Vorgehensweisen, Modelle und auch Programmcode. Als *Domäne* wird dabei im allgemeinen ein abgegrenztes Arbeits- oder Wissensgebiet, d.h. ein problemspezifischer Anwendungsbereich bezeichnet. Eine Domäne wird durch die Terminologie und die Konzepte der beteiligten Personen (Fachleute) und Systeme charakterisiert. [Simos, 1997] de-

finiert eine Domäne als einen Bereich von Funktionalitäten in einem Satz oder einer Klasse von Softwaresystemen. Der Begriff Domäne kann hierbei eine einzelne Anwendungsart beschreiben oder ein ganzes Anwendungsgebiet. Es lässt sich sowohl von einer Domäne »Taschenrechner« , wie von einer Domäne »mathematische Anwendungen«, die neben der Anwendungsart Taschenrechner auch ergänzende Produkte wie Funktionsplotter oder Formelsatzeditoren mit einbezieht, sprechen. Der Begriff der Domäne unterstützt damit insbesondere sowohl die Sicht einer Produktlinie, als auch einer Produktfamilie. Häufig ist eine Domäne auch durch den geschäftlichen Kontext einer Unternehmensaktivität vorgegeben.

Ein Domain Engineering umfasst die folgenden Phasen bzw. Prozesse:

- Die **Domänenanalyse** bestimmt die zu bearbeitende Domäne und grenzt ihren Umfang ein. Sie erfasst das vorhandene Wissen, identifiziert und analysiert die relevanten Daten, um ein kohärentes Domänenmodell zu erstellen. Durch die Analyse wird das Wissen formalisiert und weiteren Entwicklungsphasen bereitgestellt.

- Im **Domänendesign** wird aus den in der Analyse gewonnenen Erkenntnissen eine Architektur für die Systemfamilien der Domäne erstellt. Die Elemente (Komponenten) der Architektur und ihre Verwendungsmuster werden festgelegt. Zum Design gehört ein Produktionsplan. Aus ihm ist ersichtlich, wie aus der Architektur und den Komponenten das konkrete System entstehen soll.

- Zur **Domänenimplementation** gehört die Erstellung der Komponenten und die Umsetzung des Produktionsplanes. Damit werden unter Verwendung der entsprechenden Technologien die realen Anwendungssysteme implementiert.

5.3.1 Domänenanalyse

In Analogie zum architekturzentrierten Produktlinienkonzept steht auch bei einer generativen Vorgehensweise nicht die Betrachtung einer einzelnen Anwendung im Vordergrund, sodass eben die Merkmale eines gesamten Anwendungsbereichs berücksichtigt werden müssen. Der Begriff der Domäne zur Kennzeichnung eines solchen Anwendungsbereichs ist dabei allerdings noch weniger scharf festgelegt als der Begriff der Komponente. Auch hier fällt die Skalierung bzw. Granularität (Umfang des betrachteten Bereichs) sehr unterschiedlich aus.

5.3.1.1 Umfang der Domäne

Die Domänenanalyse beginnt mit der Auswahl, der Identifizierung und der Bestimmung des Umfangs der betrachteten Domäne. Die Auswahl der Domäne erfolgt anhand der zu erreichenden Ziele. Dabei sind die beteiligten Parteien, vor allem die Kunden, und deren Interessen von zentraler Bedeutung. Der Umfang kann festgelegt werden, indem Beispiele und Gegenbeispiele für Systeme angegeben werden, die zur Domäne gehören. Es können Regeln aufgestellt werden, welche Systeme oder welche Merkmale zur Domäne gehören und welche nicht. Die Abgrenzung der Domäne muss neben der Machbarkeit auch noch ökonomische Beweggründe sowie aus der Zielsetzung entstehende und weitere relevante Begrenzungen berücksichtigen.

5.3.1.2 Kategorisierung des Umfanges und der Domänen

Der Umfang (engl. *scope*) einer Domäne wird kategorisiert in horizontalem und vertikalem Umfang. Der horizontale Umfang wird auch *system category scope* genannt. Er bestimmt, wie viele verschiedene Systeme in der Domäne sind. Beispielsweise hat die Domäne der Container (Mengen, Vektoren, Listen, etc.) einen größeren horizontalen Umfang als die Domäne der Matrizen, da es mehr Applikationen gibt, die Container benötigen, als Applikationen die Matrizen benötigen. Der vertikale Umfang, auch *per-system scope* genannt, bestimmt welche (Einzel-) Teile der Systeme in der Domäne sind. Je größer die Anteile von Systemen, die in der Domäne enthalten sind, desto größer ist deren vertikaler Umfang. Beispielsweise ist der vertikale Umfang der Domäne der Container geringer als der Umfang der Domäne von Textverarbeitungen. Die Container beinhalten nur einen sehr kleinen Teil der Funktionalität einer Textverarbeitung .

Anhand dieser Kategorisierung des Umfanges wird zwischen verschiedenen Domänentypen unterschieden:

- Vertikale vs. horizontale Domänen.
- Gekapselte vs. diffuse Domänen.

Vertikale Domänen enthalten komplette Systeme. Horizontale Domänen enthalten dagegen nur Teile der Systeme. Gekapselte Domänen sind horizontale Domänen, die gleichartige Teile umfassen. Anders ausgedrückt haben die Teile eine bestimmte gleichartige Lage und Funktion innerhalb der betrachteten Systeme. Diffuse Domänen sind zwar auch horizontale Domänen, die jedoch *verschiedenartige* Teile der Systeme enthalten.

5.3.1.3 Beziehungen zwischen Domänen

Im allgemeinen werden drei Arten von Beziehungen zwischen Domänen unterschieden [Czarnecki & Eisenecker, 2000]:

- A ist in B enthalten: Das gesamte Wissen der Domäne A ist ebenfalls in Domäne B enthalten. A wird dann als *Subdomäne* von B bezeichnet. B ist damit eine *Generalisierung* von A und A ist eine *Spezialisierung* von B. Die Domäne der Karosserieproduktion ist beispielsweise eine Subdomäne der Automobilherstellung.

- A benutzt B: Das Wissen der Domäne A benutzt bzw. referenziert das Wissen der Domäne B. Das bedeutet, dass es sinnvoll ist, Aspekte der Domäne A in Ausdrücken der Domäne B zu repräsentieren. B wird damit als *Support Domain* (unterstützende oder Hilfs-Domäne) von A bezeichnet. In Anlehnung an das vorhergehende Beispiel wäre die Reifenherstellung eine Support-Domain der Automobilherstellung. Die Reifenherstellung selbst ist unabhängig vom Automobilbau, da aber in der Autoherstellung Reifen benötigt werden, werden dementsprechend die Daten der Reifen benötigt.

- A ist analog zu B: Es gibt eine beträchtliche Ähnlichkeit zwischen den beiden Domänen. Trotzdem ist es nicht unbedingt der Mühe wert, eine Domäne in den Ausdrücken und Repräsentationen der anderen Domäne zu formulieren. Die Untersuchung der einen Domäne könnte aber nützliche Fakten oder Schlußfolgerungen für die andere Domäne hervorbringen.

5.3.1.4 Fachwissen

Nach der Bestimmung des Umfanges der Domäne müssen Wissensquellen identifiziert und erfasst werden. Zu diesen Quellen gehören bereits existierende Systeme der Domäne, Fachleute, Literatur, Experimente, Anforderungen zukünftiger Systeme, Standards, Marktstudien, technologische Entwicklungen, bisherige, aktuelle und potenzielle Kunden usw.

Neben der Sammlung des Fachwissens ist deren Organisation besonders wichtig. Durch eine gute Organisation des bestehenden Fachwissens werden mögliche Schlußfolgerungen, die zu neuen Erkenntnissen führen, erleichtert oder erst ermöglicht.

5.3.1.5 Domänenmodell

Das Ziel der Domänenanalyse ist die Erstellung des Domänenmodells. Das Domänenmodell beschreibt die Gemeinsamkeiten, Unterschiede und Abhängigkeiten der Merkmale der Systeme. Es beschreibt nur die fachlich relevanten Sachverhalte und keine technisch oder anderweitig motivierten Sachverhalte ([Oestereich, 1999]).[13] [Simos, 1997] nennt es ein semantisches Modell von gemeinsamen und unterschiedlichen Merkmalen, die beim Vergleich repräsentativer Systeme auftauchen. Ein Domänenmodell ist im Allgemeinen aus folgenden Teilen aufgebaut:

- Domänendefinition: Identifizierung und Umfang der Domäne.
- Domänenlexikon: Das Vokabular der Domäne.
- Konzeptuelle Modelle: Die Konzepte der Domäne werden in passenden Modellen ausgedrückt. Dazu können beispielsweise Objektdiagramme, Zustandsübergangsdiagramme oder auch Datenflußdiagramme gehören. Zu den Modellen gehört auch eine ergänzende textuelle Beschreibung.
- Merkmalsmodelle (feature models): Merkmale (features) beschreiben Eigenschaften, die von direktem fachlichen Interesse für die Kunden oder auch Entwickler sind.[14] Sie stellen wiederverwendbare und konfigurierbare Eigenschaften der Systeme der Domäne dar. Merkmalsmodelle beschreiben welche Kombinationen von Merkmalen sinnvoll sind und welche Kombinationen unter bestimmten Bedingungen bevorzugt werden. Sie stellen damit die Konfigurationsmöglichkeiten des konzeptuellen Modells dar.

5.3.1.6 Merkmale

Die Bedeutung und der Sinn eines Merkmals bzw. eines Merkmalsmodells soll kurz an einem stark vereinfachten Beispiel der

[13] Der Autor ist daher der Ansicht, dass daher zur Repräsentation entsprechender Modelle auch technologienahe Darstellungen wie etwa die gängige UML vermieden werden sollten, da sie in einem zu frühen Stadium zuviel implementatorisches Gedankengut induzieren. Entsprechend fiel auch in der vorliegenden Arbeit die Entscheidung für den Einsatz von Merkmalsmodellen.

[14] Hierin liegt die entscheidende Abgrenzung zu anderen Eigenschaftsbegriffen wie dem Attribut oder des Property in dieser Arbeit.

Domäne Automobilbau illustriert werden: Das Konzept des Automobils beinhaltet vier Räder, einen Motor, ein Lenkrad, Sitzplätze, Karosserie und Getriebe. Die Merkmale bestimmen nun ob in das Automobil ein Otto-Motor oder ein Diesel-Motor eingebaut wird, wieviel PS er hat, ob ein 5-Gang- oder ein Automatik-Getriebe verwendet wird usw. Ein Merkmalsmodell dient dazu sinnvolle und präferierte Kombinationen von Merkmalen festzulegen. Beispielsweise wird die Kombination der Merkmale Sonnendach und gleichzeitg eine Cabriokarosserie *ausgeschlossen* sein. Die Kombination von 40PS-Dieselmotor mit Automatikgetriebe und Klimaanlage wird zu den weniger *bevorzugten* Kombinationen gehören. Die Auswahl eines Merkmals kann die Auswahlmöglichkeit anderer Merkmale beeinflussen, sodass eine Priorisierung sinnvoll sein kann (vgl. a. Abschnitt 6.4.3).

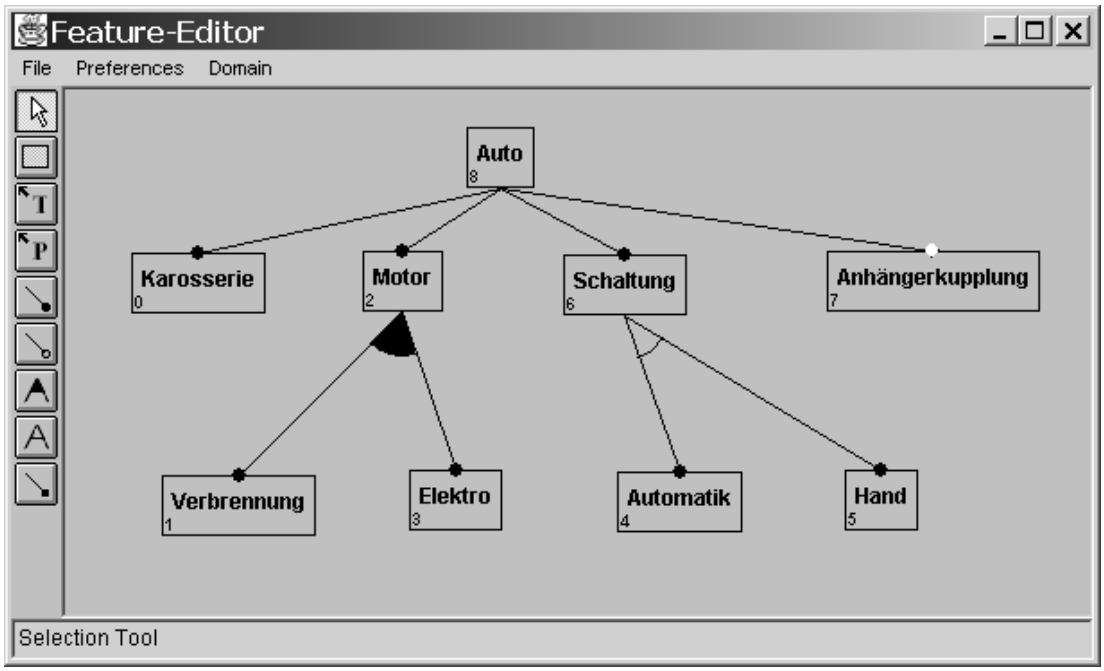


Abbildung 5.1
Screenshot des Merkmalseditors

Merkmale und Merkmalsmodelle werden eingesetzt, um Gemeinsamkeiten und Variationen der Systeme der Domäne auszudrücken. Diese Konzepte sind durch die *Feature Oriented Domain Analysis* (FODA) eingeführt worden (s. etwa [Schnell et al., 1996] zur Übersicht). In FODA wird ein Merkmal als eine variable Eigenschaft des Systems, die für den Endbenutzer sichtbar ist, definiert. Hier soll eine etwas erweiterte Definition verwendet werden: Ein Merkmal ist eine ausgezeichnete Eigenschaft des Sy-

stems, die relevant für an dem System beteiligte Parteien ist. Damit sind auch die Entwickler mit einbezogen und Eigenschaften, die beispielsweise den Kostenfaktor ändern, aber vom Endanwender nicht wahrgenommen werden. Das FODA Merkmalsmodell (feature model) besteht aus den folgenden Elementen:

- Merkmalsdiagramm – Das Merkmalsdiagramm ist eine hierarchische Zusammenstellung der Merkmale. Dort ist beschrieben, welche Merkmale unabdingbar, alternativ oder optional sind. Abbildung 5.1 illustriert ein Beispiel (vgl. a. Abschnitt 6.4.3).
- Merkmalsdefinitionen – Sie beschreiben die Merkmale und legen den Zeitpunkt fest, wann ein Merkmal bestimmt wird. Das kann bei der Compilierung, der Aktivierung oder zur Laufzeit, aber auch zu anderen Zeitpunkten geschehen.
- Kombinationsregeln für die Zusammenstellung von Merkmalen – Die Regeln legen fest, welche Kombinationen von Merkmalen gültig sind und welche nicht.
- Einsatzgründe für Merkmale – Sie schlagen vor und begründen, unter welchen Bedingungen ein Merkmal eingesetzt werden sollte und wann nicht.

FODA unterscheidet die Merkmale nach ihrem Zweck, in kontextuelle, repräsentative und operationale Merkmale. Kontextuelle Merkmale beschreiben das Ziel oder den Verwendungszweck der Applikation. Dazu gehören auch Merkmale wie Genauigkeit und Leistungsfähigkeit. Repräsentative Merkmale beschreiben die Darstellung der Informationen für den Benutzer oder für weiterverarbeitende Applikationen. Operationale Merkmale geben die eigentlichen Funktionen der Applikation an.

Merkmale und Merkmalsmodelle beschreiben die Konfigurationsmöglichkeiten der zu erstellenden Systeme. Sie dienen als Basis für das Design einer Produktfamilie und zur Erstellung der einzelnen Komponenten, die diese Merkmale tragen. Sie beinhalten das Konfigurationswissen, um die Erstellung von Applikationen zu automatisieren. Ebenso bieten sie damit einen Katalog von Konfigurationen, die dem Kunden angeboten werden können.

5.3.2 Domänendesign

Das Domänendesign hat die Erstellung einer Architektur für die Systeme der Domäne und den Aufbau eines Produktionsplanes zum Gegenstand.

5.3.2.1 Architektur

Eine Softwarearchitektur besteht aus den Beschreibungen der einzelnen Komponenten der Architektur und deren Verknüpfungen bzw. Interaktionen. Für die Zusammenstellung der Komponenten und deren Verknüpfungen kann es Restriktionen und / oder Regeln geben. Die Architektur hat im allgemeinen Fall verschiedene Sichtweisen, die entsprechende Modelle erfordern, wie zum Beispiel Klassen-, Ablauf- und Zustandsübergangsdiagramme.[15] Ein Softwarearchitekten soll rasch erkennen können, wie das System aufgebaut ist und an welchen Stellen bestimmte Funktionen oder Konzepte umgesetzt wurden (vgl. a. das Konzept der Entwurfsmuster [Gamma et al., 1994]).

Die Architekturen lassen sich in starre und flexible Architekturen unterteilen. Starre Architekturen haben eine festgelegte nicht änderbare Struktur, in der die Variationspunkte (vgl. [Griffel, 1998]) festgelegt sind und damit die Komponenten festgelegte Schnittstellen erfüllen müssen, um eingesetzt zu werden. Flexible Architekturen erlauben strukturelle Veränderungen. Damit kann selbst die Architektur des Systems konfiguriert werden und den Domänenanforderungen angepasst werden. Das kann zum einen von Nutzen sein, wenn die Domäne geändert oder erweitert wird, oder eine Domäne mit stark diversifizierten Anwendungssystemen vorliegt. Die vorliegende Arbeit betrachtet in diesem Sinne ausschließlich starre Architekturen.

5.3.2.2 Produktionsplan

Der Produktionsplan beschreibt, wie das konkrete System aus den Komponenten und der Architektur erstellt wird. Darunter fallen auch die Messung, die Verfolgung und die Optimierung des Produktionsprozesses. Dem Kunden bietet der Produktionsplan die Wahl der Ausprägungen des zu erstellenden Systems.

Für die Erstellung des konkreten Systems wird zwischen drei Automatisierungsstufen unterschieden [Czarnecki & Eisenecker, 2000].

- Manuelle Fertigung: Der Architektur und den Komponenten wird ein Entwicklungshandbuch beigelegt. Die Applika-

[15] Der Ansatz dieser Arbeit geht von einer stärkeren Trennung dieser Einzelaspekte aus und grenzt daher den Architekturbegriff auf rein *strukturelle* Aspekte ein. Die dahinterstehende Idee ist, dass die einzelnen, dann in einer Struktur angeordneten Komponenten selbst die ensprechende Verknüpfungsinformation für ihr Interaktionspotenzial liefern (vgl. Kapitel 4).

tionen müssen mit den Komponenten manuell erstellt werden.[16]

- Teilautomatisierte Fertigung: Hierbei wird die Verknüpfung der Komponenten und auch eventuell weiteren Teilen des Fertigungsprozesses von Entwicklungswerkzeugen unterstützt. Eine Framework-basierte Entwicklung ist typischerweise hier angesiedelt.

- Vollautomatische Fertigung: Hierbei kann ein Kunde seine Wunschkonfiguration der zu fertigenden Applikation festlegen und die Applikation kann dann vollautomatisch mit Generatoren anhand der vorgegebenen Konfiguration erstellt werden. Dieses Ziel verfolgt die Generative Programmierung und Konstruktion von Anwendungen.

5.3.3 Domänenimplementation

Die Domänenimplementation beinhaltet die konkrete Implementation im Rahmen der Architektur, der Komponenten und des Produktionsplanes. Dazu gehört die Auswahl der passenden Technologien, die Erstellung von Entwicklungshandbüchern, die Implementation von Konfigurationsassistenten und eventuell Generatoren und der Aufbau der nötigen Infrastruktur.

Das Erstellen der konkreten Applikation aus den Ergebnissen des Domain Engineering wird als *Application Engineering* bezeichnet. Anforderungen, die nicht im Domänenmodell berücksichtigt wurden, gelten als Spezialerweiterungen und müssen separat erstellt werden. Das Domain Engineering wird als kontinuierlicher Prozess betrachtet. Das bedeutet, dass die einzelnen Phasen kontinuierlich weiterentwickelt werden, insbesondere da aus dem Application Engineering Erfahrungen in das Domain Engineering zurück fließen.

5.3.4 Feature Oriented Domain Analysis (FODA)

FODA ist eine der ausgereiftesten und am besten dokumentierten Methoden zur Domänenanalyse. Sie ist vom Software Engineering Institute (SEI) der Carnegie Mellon Universität entwickelt worden. Ihr spezieller Charakter rührt aus der Betonung

[16] Die Applikationshandbücher und »Cookbooks« integrierter Schaltkreise (ICs) als Paradebeispiel einer erfolgreichen Komponentensicht bieten eine passende Analogie.

von Merkmalen und deren Modellierung. Der FODA Prozess besteht aus den Phasen Kontextanalyse und Domänenmodellierung.

Die Kontextanalyse definiert die Grenzen und den Umfang der zu analysierenden Domäne. Der Umfang der Domäne wird so festgelegt, dass nützliche Produkte für die Domäne erstellt werden können. Die Grenzen der Domäne hängen vom Fachwissen und den Grenzen des jeweiligen Projektes ab. Des weiteren werden Beziehungen zu anderen Domänen oder Einheiten aufgezeigt.

Die Modellierungsphase identifiziert die Gemeinsamkeiten und Unterschiede der Applikationen der Domäne. Dazu gehören die folgenden Schritte:

- Informationsanalyse – Die Objekte der Domäne und ihre Beziehungen untereinander werden aus den vorliegenden Informationen identifiziert. Zur Modellierung können beispielsweise Objekt-, Entity-Relationship-Modelle oder semantische Netze herangezogen werden. Ziel dieser Analyse ist ein *Informationsmodell*, das mit dem oben beschriebenen konzeptuellen Modell korrespondiert.[17]

- Merkmalsanalyse (engl. *feature analysis*) – Die Merkmale werden identifiziert und ein Merkmalsmodell wird aufgebaut.

- Operationale Analyse – sie identifiziert die Gemeinsamkeiten und Unterschiede der Steuerung und der Datenflüsse der Domäne. Ziel ist ein operationales Modell. Es beschreibt die Funktionsweise der Applikation, indem die Objekte des Informationsmodels mit den Merkmalen im Merkmalsmodell in Beziehung gesetzt werden.

Ein weiteres Produkt der Modellierungsphase ist das Domänenwörterbuch (engl. *domain dictionary*). In ihm wird die Terminologie der Domäne festgehalten.

Neben FODA existieren noch diverse weitere Methoden zur Domänenanalyse und zum Domain Engineering allgemein. Dazu gehören etwa das Organization Domain Modeling (ODM), Draco, Capture und DARE um nur einige zu nennen. Die verschiedenen Methoden des Domain Engineering sind von terminologischen Unterschieden abgesehen relativ ähnlich. Der Prozessablauf ähnelt sich stark. Die Unterschiede liegen in den konkreten

[17] vgl. a. die »Viewpoints« des RM-ODP Modells [Commission, 1995]

Modellierungstechniken, Notationen, Methoden der Softwarearchitektur und den Implementationstechniken. Eine Zusammenfassung solcher Methoden ist in [Czarnecki & Eisenecker, 2000] zu finden. Der Prozess der generativen Softwarekonstruktion in dieser Arbeit baut auf der FODA-Vorgehensweise auf, da diese am stärksten die Variationsfähigkeit eines System aufrecht erhält und ihr Merkmalsbegriff sowie ihre operationalen Modelle direkt dem hier zu Grunde gelegten Komponenten- bzw. Typmodell zugeordnet werden können. Die homogene Repräsentation sowohl von Modellspezifikationen bzw. Randbedingungen (Constraints), Komponenteneigenschaften und Merkmalsdiagrammen in Form von Merkmalsstrukturen erlaubt zudem eine elegante Implementation und entsprechend assoziierte Verarbeitung (vgl. Kapitel 6).

5.3.5 Fazit

Das Domain Engineering hat das Ziel einen Produktionsplan für die Erstellung von Applikationen einer Domäne aufzubauen, sodass deren allgemeine Prinzipien und häufig (mehrmals) auftretende Systemanteile (Komponenten) im Vordergrund stehen. Im Gegensatz dazu steht das klassische Software Engineering, mit dem Ziel eine spezielle Applikation zu erstellen. Im Domain Engineering wird die Wiederverwendung von Analyse und Designmodellen verfolgt und weniger die Wiederverwendung von Programmcode. Der Aufwand für das Domain Engineering ist dementsprechend deutlich höher als im Rahmen des heutigen Software Engineering häufig üblich. Der Vorteil der umfangreicheren »Vorarbeit« liegt jedoch in der Möglichkeit, flexibel und schnell verschiedene Applikationen für die Domäne erstellen zu können. Das lohnt sich natürlich nur dann, wenn es auch einen Bedarf für verschiedenartige Systeme in der Domäne gibt. Ein Vergleich mit der industriellen Produktion von Gütern im Gegensatz zur Einzelferigung bietet sich hier an. In der industriellen Produktion werden Güter mit definierten Variationspunkten in Massenproduktion gefertigt. Durch diese Massenproduktion wird das Einzelstück erst günstig, im Gegensatz zur Einzelfertigung. Durch den Einsatz einer vollautomatischen Generierung von Applikationen wird das Ziel der »industriellen« Produktion von Software angenähert, wobei ein Generator für die Auswahl der richtigen Komponenten und deren Komposition sorgt [Batory & Geraci, 1997]. Dies geschieht im Rahmen einer vorgebenen *Anwendungsstruktur* (der Ausprägung einer Architektur,

vgl. a. [Clements & Northrop, 1996]). Genau dieser Ansatz wird im Weiteren verfolgt.

5.4 Software-Generatoren

Generatoren können für die automatische Erstellung unterschiedlicher Anwendungen einer Domäne eingesetzt werden, indem sie die Ergebnisse des Domain Engineerings verarbeiten. Generatoren nehmen dem Programmierer dabei häufig vorkommende bzw. sich wiederholende Aufgaben ab und sorgen für eine korrekt zusammengesetzte Anwendung. Mit Hilfe eines Generators kann die mit Mitteln des Domänenmodells ausgedrückte Spezifikation einer Anwendung in ein ausführbares Programm umgewandelt werden, unter Aufrechterhaltung der technologischen Abstraktion – das Wissen um solche Details bleibt im Generator verborgen. Generatoren lassen sich als eine Art Compiler für domänenspezifische Sprachen ansehen, die einen höheren Grad der Abstraktion von der eigentlichen Implementierung als es eine übliche Programmiersprache erlauben. Eine gute Diskussion findet sich in [Smaragdakis & Batory, 2000] und soll hier nicht geführt werden.

Bei der Erstellung des Domänenmodells muss also eine Beschreibungssprache für den Problemraum gefunden werden. Der Programmierer muss nur noch den Problemraum beschreiben, nicht den Lösungsraum mit den technischen Details der konkreten Umsetzung in einer Programmiersprache. Der Problemraum wird auf diese Weise vom Lösungsraum getrennt [Czarnecki & Eisenecker, 2000]. Das Problem wird in der Fachsprache der Anwendung selbst spezifiziert. Die Sprache ist damit präziser auf das eigentliche Problem zugeschnitten. Der Generator muss aus diesen Ausdrucksmitteln ableiten können, welche Softwarebausteine wie benutzt werden müssen, um die gewünschte Anwendung zu erhalten. Er muss die Komponenten miteinander verknüpfen und letztlich ein ausführbares Programm erstellen. Für diese Umsetzung können verschiedene Techniken eingesetzt werden. Es fließen hier Konzepte generischer, transformativer und generativer Techniken ein ([Neubert, 1998]).

5.4.1 Motivation für Generatoren

Der Einsatz von Generatoren verzögert konkrete Implementationsentscheidungen. Die Umsetzung der Problembeschreibung in

eine verwendete Technik und Programmierung kann zu einem beliebigen Zeitpunkt erfolgen. Problem- und Lösungsraum sind voneinander getrennt und können damit auch getrennt voneinander betrachtet werden. Ein häufiges Problem der Softwareentwicklung ist, dass Programmierer und Kunden unterschiedliche »Sprachen« sprechen. Der Kunde ist technische Fachausdrücke der Entwicklung nicht gewohnt und der Programmierer kennt das Denken und die Fachterminologie einer Anwendungsdomäne nicht. Der Einsatz von Generatoren erlaubt eine Trennung von einerseits technisch orientierten Programmierern (diese erstellen Komponenten) und fachlich orientierten Entwicklern (diese erstellen Anwendungen) – letztere sollen hier als Software*konstrukteure* bezeichnet werden. Die entsprechende Abstraktion in der Entwicklung illustriert die Abbildung 5.2 aus [Clements, 1996].

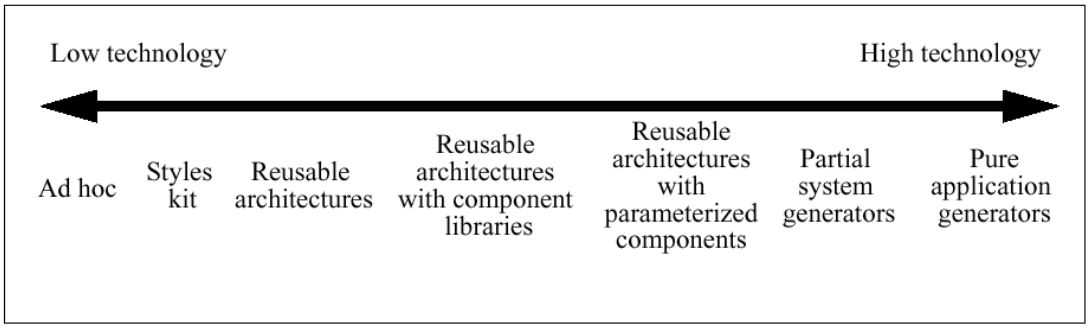


Abbildung 5.2
Spektrum der Entwicklungsansätze

Es ist möglich, Quellcode für verschiedene Programmiersprachen oder Anwendungsumgebungen zu generieren. Es ist zum Beispiel ein Generator denkbar, der eine Anwendung für eine CORBA-Umgebung zusammenstellt und ein anderer, der Anwendungen auf Basis des COM+-Modells von Microsoft zusammenstellt – unter Verwendung derselben Architekurspezifikation. Die Beschreibung des Problemraumes als Grundlage der Generierung ändert sich dabei also nicht. Der zu erzeugende Code kann allerdings nur durch einen erneuten Generatordurchlauf an neue Gegebenheiten angepasst werden, was allerdings im Vergleich zu einer manuellen Anpassung rasch durchzuführen ist. Dies ist auch einer der Gründe, warum der Autor in der Kombination der Kompontensicht mit einem generativen Vorgehen wie es diese Arbeit propagiert, eine ideale Entwicklungsweise sieht. Die Komponentensicht fördert die explizite Konfiguration der Einzelbausteine sowie deren lose Kopplung, sodass auch generierte,

aus Bausteinen zusammengesetzte Anwendungen im Nachhinein noch aktualisiert oder adaptiert werden können.

Die manuelle Erstellung von Quellcode ist zeitaufwendig und fehleranfällig. Einfache, sich wiederholende Aufgaben müssen nicht von Experten vorgenommen werden, sondern können sehr viel schneller und sicherer automatisch erfolgen. Automatisch generierte Programme besitzen bessere Voraussetzungen zur Etablierung einer (hohen) Softwarequalität, da sie aus einem besser messbaren Grundgerüst auf stets gleiche Weise abgeleitet werden. Der Generator muss sicherstellen, dass die verwendeten Bausteine zueinander kompatibel sind und die Kombination eine auch semantisch korrekte Anwendung ergibt. Auf diese Weise kann eine *correctness by construction* erreicht werden.

[Smaragdakis & Batory, 2000] nennt noch die erweiterte Fehlersuche als Vorteil, die automatisch durch das vorhandene Domänenwissen unterstützt werden kann. Das Domänenwissen kann auch eingesetzt werden, um domänenspezifische Optimierungen durchzuführen. Ein Generator könnte zum Beispiel den für die zu generierende Anwendung optimalen Suchalgorithmus auf Grund der zu bearbeitenden Daten bestimmen – [Neubert, 1998] liefert etwa entsprechende Beispiele.

5.4.2 Produktkonfiguration (Customizing)

Durch den Einsatz von Generatoren ist eine konsistente Handhabung einer grösseren Variantenvielfalt möglich. Der Programmierer kann durch Angabe der gewünschten Eigenschaften genau die Software beschreiben, die er erhalten möchte und so sein Produkt auf einen Kunden zuschneiden. Aus einer Vielzahl von vorhandenen Softwarebausteinen kann der Generator diese Anwendung dann redundanzfrei zusammenstellen, d.h., das »organische Wachstum« (oder besser »Wuchern«) komplexer Software[18] wird verhindert. Generell lassen sich heute zwei Extreme einer Software unterscheiden:

- Standardsoftware – Eine Standardsoftware wie Microsoft Word bietet viele Funktionalitäten, um ein möglichst großes Anwendungsgebiet abzudecken. Ein Großteil der Funktionalität wird vom Anwender aber gar nicht benötigt oder nicht genutzt. Bei der Installation kann nur sehr begrenzt die gewünschte Funktionalität reduziert werden. Es bleibt ein großer Teil an Overhead, der Ressourcen benötigt und die Programmbedienung unnötig verkompliziert.

[18] Das Microsoft Office-Paket ist ein bekanntes Paradebeispiel.

- Individualsoftware – Um eine Anwendung zu erhalten, die genau den Anforderungen entspricht und keinen unnötigen Overhead besitzt, muss eine Software individuell programmiert werden. Individualsoftware ist sehr teuer und zeitaufwendig zu erstellen, da sie nur in einem eng begrenzten Bereich zum Einsatz kommt. Ihre Spezialisierung erschwert eine spätere Erweiterung auf anderen Einsatzbereiche.

Der Einsatz von Generatoren liefert in Kombination mit einer Komponentensicht die Möglichkeit, einen fließenden Übergang beider Ansätze zu ermöglichen. Individuelle Software kann aus Standardbausteinen rasch zusammengestellt werden. Es werden nur wirklich die Bausteine verwendet, die für die gewünschte Anwendung benötigt werden. Ungewollte bzw. nicht benötigte Funktionalität wird drastisch reduziert. Sollten sich die Anforderungen ändern und eine neue Produktvariante erforderlich sein, kann eine neue Anwendung mit entsprechender Funktionalität generiert werden. Somit ist insbesondere eine sehr schnelle Reaktion auf Änderungswünsche der Kunden bzw. Endanwender möglich, denen gleichzeitig dennoch eine für sie spezifische, eventuell sogar optimale Lösung geboten werden kann. Dieses verlockende Potenzial einer solchen Vorgehensweise stellt die Motivation für die nachfolgende genauere Betrachtung generativer und verwandter Techniken dar.

5.5 Generative Programmierung und verwandte Konzepte

Im Weiteren soll das allgemeine Konzept der generativen Programmierung betrachtet und andere, auf diesen Ansatz einflussnehmende Ideen als Grundlage der Konzeption bzw. Abgrenzung eines eigenen Ansatzes vorgestellt werden. Allen Ansätzen liegt heute das Prinzip der »Trennung der Belange« (engl. *Separation of Concerns*) zu Grunde. Durch einen Generator bzw. einem Weaver bei der Aspektorientierung (s.u.) oder ähnlichen Compiler-artigen Mechanismen werden die getrennt betrachteten Belange wieder in Relation zueinander gebracht. Die in der Literatur bislang zu findenden im Folgenden vorgestellten Konzepte erzeugen dabei »traditionellen« Quellcode.

Es werden im Folgenden Techniken wie die Aspektorientierte Programmierung und die verwandte Adaptive Programmierung vorgestellt. Viele Analogien zur generativen Programmie-

rung haben auch Transformationssysteme. Einen interessanten Ansatz stellt dabei das sogenannte *Intentional Programming* von Microsoft dar, das vollständig von einer einzelnen Programmiersprache abstrahiert, indem ein Entwickler bei der Programmerstellung seine Absichten in sehr abstrakter Form ausdrücken kann, die anschließend in typische programmiersprachliche Konstrukte transformiert wird.

5.5.1 Aspektorientierte Programmierung (AOP)

Die AOP [Kiczales et al., 1997] betont bereits durch ihre Bezeichnung die Idee, ein zu erstellendes Gesamtsystem während der Entwicklung aus jeweils verschiedenen und getrennten Blickwinkeln zu betrachten und so stets eine Konzentration auf einzelne *Aspekte* des Systems zu erreichen. Dieser Grundsatz wird als *Separation of Concerns* bezeichnet.[19] Programme werden in Form von lose gekoppelten, miteinander kooperierenden Einheiten ausgedrückt, von denen jede nur die Aspekte eines bestimmten Kontextes beschreibt. Die Aspekte werden einzeln spezifiziert und durch einen Weaver (dt. *Weber*) zu einem Programmcode verarbeitet (»verwebt«), der im Ergebnis einer normalen Programmierung entspricht. Der erzeugte Quellcode sollte dabei mindestens genauso gut sein, wie der manuell geschriebene. Änderungen können einfach an den einzelnen Aspekten vorgenommen werden, die dann neu zusammengewoben werden müssen. Beispiele für typische Aspekte einer Softwareapplikation sind etwa Marshaling, Synchronisation, Fehlerbehandlung, Quality-of-Service und die verwendeten Datenstrukturen. Durch die Entwicklung und Verwendung von speziellen Sprachen für häufig auftretende Aspekte, können diese besonders verständlich und einfach ausgedrückt werden. Die Anpassbarkeit der Programme wird auch erhöht, weil Änderungen jeweils zentral an einer Stelle vorgenommen werden können. Die Idee der Aspektorientierung und entsprechende Werkzeuge entstanden am Xerox Palo Alto Research Center.[20]

[19] Siehe allerdings auch zur Wechselwirkung der Aspekte etwa [Klein et al., 1998]. Interessante Betrachtungen zu einer *mehrdimensionalen* Trennung der Belange findet sich auch im Rahmen von HyperJ `http://www.alphaworks.ibm.com/`.

[20] `http://www.parc.alto.com/aop`

5.5.1.1 Der Begriff des Aspekts

Die AOP arbeitet mit zwei wichtigen Konstrukten: dem Aspekt und der Komponente. In [Lieberherr, 1997] werden Aspekte und Komponenten wie folgt definiert:

With respect to a system and its implementation using a generalized procedure language, a property that must be implemented is (1) a component, if its implementation can be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API). By cleanly, we mean well-localized, and easily accessed and composed as necessary or (2) an aspect, if its implementation cannot be cleanly encapsulated in a generalized procedure

Eine Komponente ist in der AOP also ein Programmteil, wobei dies ein Objekt, eine Methode oder ähnliches sein kann, der abgegrenzt werden kann vom Rest der Programmierung. Ein Aspekt dagegen betrifft mehrere Bereiche eines Programmsystems, so das er nicht eindeutig herausgetrennt werden kann, sondern systemübergreifend betrachtet werden muss.[21] Eine häufig zu findende Analogie ist etwa die Bauzeichung eines Hauses (Gesamtsystem). Eine solche Zeichnung enthält die gesamte Information zur Systemerstellung (Hausbau), allerdings separat gekennzeichnet für die jeweils betroffenen Rollen (Handwerker, für die ein bestimmter Aspekt von Belang ist). So sind sanitäre Anlagen und Leitungen beispielsweise blau markiert und elektrische Leitungen rot, während der Grundriss für den Architekten bzw. Maurer schwarz erscheint. Auf diese Weise sehen Klempner, Elektriker und Maurer in erster Linie die sie betreffenden Systemteile. Dennoch entsteht insgesamt ein Haus.

Es gibt zwei Arten von Aspekten. Vertikale Aspekte beziehen sich auf die Art der Anwendung. Ein vertikaler Aspekt wäre zum Beispiel eine Bilanzerstellung. Horizontale Aspekte haben Auswirkungen auf viele Teilbereiche des Programmes. Darunter fallen etwa Sicherheitsaspekte oder Synchronisationsaspekte.

5.5.1.2 Motivation der Aspektorientierung

Die Anpassbarkeit und damit die Wieder- bzw. Weiterverwendbarkeit von Programmen ist wie bereits betont eine der grössten Herausforderungen im Rahmen einer Industrialisierung der Softwareentwicklung. Änderungen eines Details können zu Änderungen in vielen Programmteilen führen. Bei der Aspektorientierung müssen Änderungen konzeptionell nur an einer ein-

[21] Dies wird auch als *cross-cut* Eigenschaft bezeichnet.

zigen Stelle, bei dem entsprechenden Aspekt der Programmierung, vorgenommen werden.[22] Der Code für eine Synchronisation ist so beispielsweise nicht mit sequentiellen Code vermischt. Der Funktionalitätsaspekt kann dann wiederum ohne Rücksichtnahme auf die Synchronisation formuliert werden (vgl. a. [Griffel, 1998]). Der in einer speziell auf die Anforderungen und Merkmale der Synchronisation zugeschnittenen Sprache formulierte Aspekt ist einfach lokalisierbar. Die Evolution eines Programms wird hierdurch vereinfacht. Erst nach dem »Verweben« vermischen sich die systemübergreifenden Aspekte zum Gesamtsystem. Hier ist bereits eine Analogie zu einem generativen Vorgehen erkennbar: Der »Weber« ist nichts anders als ein spezieller Generator. Ein weiteres Leitbild der AOP ist die Vermeidung einer Überspezifikation. Programmcode enthält wie oben angemerkt häufig redundante Informationen. Auch hier wird die Information zusammengefasst und erst durch den Weaver auf verschiedene Programmbereiche verteilt.

5.5.2 Adaptive Programmierung (AP)

Eine spezifische Ausprägung der Aspektorientierung findet sich mit der »Adaptiven Progammierung« (AP). Die adaptive Programmierung nutzt eine zusätzliche Einschränkung durch die Verwendung sogenannter Strategiegraphen. Die Idee ist dabei zum einen wieder die Aufteilung eines Systems in einzelne Aspekte. Zum anderen werden jedoch auch die *Struktur*[23] der Bausteine und die *Navigation*[24] durch diese Struktur als getrennt vorliegende Aspekte aufgefasst. Weitere Aspekte können ohne Beachtung späterer Strukturen entwickelt werden (*structure-shy*, [Lieberherr, 1996]). Ein Weaver erstellt auch in diesem Ansatz dann aus den einzelnen Aspekten unter Berücksichtigung einer problemspezifischen Navigation durch die möglichen Strukturen einen damit redundanzfreien Programmcode.[25] Eine konkrete Umsetzung der adaptiven Programmierung

[22] Dies ist prinzipiell auch die Argumentation der klassischen Objektorientierung. Die AOP abstrahiert allerdings stärker, indem sie Aspekte nicht direkt als Programmkonstrukte vorsieht – wie es in der Regel in einer Objektsicht der Fall ist.

[23] die in Form von UML-Diagrammen oder ähnlichen Notationen vorliegt

[24] die als Strategiegraph bezeichnet wird

[25] Dass eine solche Traversierung auch auf Basis von Meta-Protokollen möglich ist, die dem Ansatz dieser Arbeit näher kommen, zeigt etwa [Batory & Geraci,].

ist »Demeter«, entwickelt in der Gruppe um Karl Lieberherr an der Northeastern University.[26] Die (adaptive) Programmierung erfolgt dabei in einer Kombination einer traditionellen Programmiersprache (etwa Java oder C++), einer speziellen Sprache zur Formulierung der Navigation innerhalb der Strukturen und in speziellen Sprachen für einzelne Aspekte. Die Grundlage des Konzepts bildet das »Gesetz von Demeter«, das die Gruppe 1987 postulierte und stilistische Regeln für ein objektorientiertes Design festlegt:

Each unit should have only limited knowledge about other units: only units »closely« related to the current unit

Danach sollten Methoden dann zum Beispiel so unabhängig wie möglich implementiert werden, d.h. unabhängig vom Aufrufer der Methode. Eigentlich sollten Methoden daher nicht einmal Rückgabewerte liefern (der Aufrufer wäre dann ja im aufrufenden Kontext von einer externen Quelle abhängig), wobei bei Demeter eher eine abgemilderte Form dieser Forderung umgesetzt wird. Durch die Anwendung des Visitor-Patterns [Gamma et al., 1994] wird diese eigene Forderung teilweise wieder aufgehoben, da in den Strategiegraphen, wenn auch eingeschränkt, Strukturwissen vorhanden ist. Die Komponentenbasierte Softwareentwicklung betont dagegen gerade konsequent die Entkopplung, die im Gesetz von Demeter gefordert wird.

Demeter fordert die Entwicklung von strukturunabhängigen Programmen, d.h. die einzelnen Teile sollten möglichst wenig Kenntnis von anderen Teilen der Programmierung benötigen. Funktionalität und Struktur sollen ähnlich der Forderung des Komponentenparadigmas jeweils für sich betont, aber eben klar getrennt werden (»structure-shy« und »function-blind«). Demeter löst dieses Problem durch die Einführung der erwähnten Strategiegraphen. Strategiegraphen bilden dabei zusammen mit einem funktionalen bzw. algorithmischen Anteil ein sogenanntes Propagationsmuster. Die Strategiegraphen markieren dabei die jeweils problemspezifischen Ausschnitte aus Klassendiagrammen. Es kann damit zum Beispiel ausgedrückt werden, welcher Pfad bei der Anwendung des Propagationsmusters in einem Klassendiagramm benutzt werden sollte, wobei alle Klassen auf diesem Pfad zu berücksichtigen und dem Algorithmusteil entsprechend anzupasssen sind (etwa das Einfügen bestimmter Methoden). Der Pfad kann durch Start- und Zielklassen (*from* ... *to* ...), sowie benötigte (*through* ...) oder auszulassende Klassen (*bypassing* ...) beschrieben werden (Redundanzfreiheit). Da nur die wirk-

[26] `http://www.ccs.neu.edu/research/demeter`

lich markanten und wichtigen Klassen genannt werden, wird eine Überspezifikation vermieden, die die Wiederverwendbarkeit der Programmteile erschweren würde. Der Algorithmusteil beschreibt die eigentliche Funktionalität, die hinzugefügt werden soll, in der jeweiligen Programmiersprache.

Insgesamt ist damit ein adaptives Programm durch mehrere unterschiedliche Aspekte gekennzeichnet:[27]

- Funktionalität – die Funktionalität wird durch *Visitors* ausgedrückt
- Struktur – wird durch UML-Klassendiagramme beschrieben
- Navigation – werden durch Strategien ausgedrückt
- Objektbeschreibung – Grammatiken (kommentierte Klassendiagramme)
- weitere Aspekte wie Synchronisation und Kommunikation

Zur Spezifikation der Synchronisation wurde die Sprache *Cool* entwickelt. Die Aspektsbeschreibungssprache für die Kommunikation ist in Entwicklung. Mit Hilfe eines Weavers werden Aspektbeschreibungen, Klassendiagramme, Strategiegraphen und Funktionalitätsbeschreibungen in ein ausführbares Programm umgewandelt.

5.5.3 Bewertung von AOP und AP

Ein Vorteil des Demeter-Ansatzes ist sicherlich die theoretisch bewiesene Korrektheit ([Lieberherr, 1997]). Aber auch in der Praxis soll Demeter die Evolution einer Software durch die Aufteilung in Aspekte vereinfachen. Es gibt mit Werkzeugen wie AspectJ und Demeter/Java Beispiele, wie die zunächst akademischen Ansätze AOP, bzw. AP der Praxis zugänglich gemacht werden kann. In der wirklichen Entwicklung kann es jedoch beliebig komplex sein, ein *weaving* zuvor getrennt identifizierter Aspekte zu erreichen. Es stellt sich die Frage, ob die Werkzeuge nur für wohlgeformte, kleine Beispiele geeignet sind. Es muss sich zeigen, ob ein Weaver in praxisnahen Beispielen in der Lage ist, komplexe horizontale, systemdurchdringende Aspekte auf

[27] Im Vergleich zum hier vorgestellten generativen Ansatz ließen sich folgenden Analogien zur Demeter-Terminologie ziehen: *operation*-Teil – Komponententyp, *navigation*-Teil – Verknüpfung durch Generator, *wrapper*-Teil: Komponenten.

Programiersprachebene zu verweben. Ein Verweben von vertikalen Aspekten sollte dagegen weniger Probleme bereiten. Sowohl AOP, als auch AP sind Zeilen- bzw. Quellcode-orieniert, d.h., sie gehen davon aus, dass alle Programmteile neu erschaffen werden müssen. Sie greifen nicht auf vorgefertigte Komponenten zurück. Demeter widerspricht wie angesprochen mit dem Visitor-Pattern eigentlich dem eigenen Gesetz von Demeter. Insgesamt ist der Autor der Ansicht, dass die Einführung getrennter jeweils problem- (bzw. aspekt-) spezifischer Sprachen neben Akzeptanzproblemen seitens der Entwickler eher eine Verlagerung denn einer Lösung der Probleme einer »zeilenweisen« Programmierung grosser Anwendungssysteme entsprechen. Wünschenswerter wäre die vollständige Abstraktion von dieser »Zeilenbene« durch den Einsatz von Black-box-Komponenten.[28]

Die Aspektorientierung stellt einen zur hier propagierten generativen Softwarekonstruktion komplementären Ansatz dar: Während letztere vorhandene Komponenten voraussetzt, werden Aspekte erst identifiziert und dann implementiert und verwoben. Sie können dabei bei der Erstellung *von* Komponenten hilfreich sein, benötigen aber komplexes Wissen bei der Erstellung einer Anwendung und sind eng mit dem Programmcode verknüpft.[29]

Ansätze wie [Smaragdakis, 1999] tragen nach Meinung des Autors zu Unrecht den Titel »large-scale«, da sie sich letztlich mit der Zeilenebene auseinandersetzen und nicht wirklich die Systemkomposition auf hoher Granularitätsebene betrachten. Wenn daher nur die weit »verteilte« Durchdringung eines System von seinen Einzelaspekten gemeint ist, sollten entsprechend weniger irreführende Begriffe gewählt werden, beispielsweise den der »Reichweite«.

Aspekte könnten auch Konfigurationspunkten gleichgesetzt werden [Griffel, 1998]. Der »cross-cut« Effekt der AOP entspricht im hier vorgestellten Ansatz dann einem übergreifenden (bei gleichzeitig lose gekoppelten Komponenten) Policy-Management über alle Konfigurationspunkte (aller beteiligten Komponenten), vergleiche auch Abschnitt 3.10 und 5.10.2. Solche *Attribut-basierten Architekturstile* sollen hier allerdings nicht weiter ver-

[28] Deren Entstehung selbstverständlich von den genannten Vorgehensweisen profitieren kann.

[29] Vgl. etwa a. Komponenten mit den in [Smaragdakis, 1999] diskutierten »Mixin Layers« : Beide sind bereits mit speziellen Funktionen ausgestattet, Mixins können durch Zuordnung einer Superklasse weitere, allgemeiner Funktionen nutzen, Komponenten können dies durch gegenseitige Nutzung ihrer Dienste (entspricht dem Kollaborationsgedanken) [Griffel, 1998].

tieft werden, für eine entsprechende Diskussion siehe etwa [Klein & Kazman, 1999].

5.5.4 Aspektorientierung vs. generative Programmierung

Zwei der häufigsten Probleme der traditionellen Programmierung sind die Anpassung einer Anwendung an geänderte Anforderungen sowie die Beurteilung einer möglichen Wiederverwendung bestehender Lösungen. Die Objektorientierung hat durch Mechanismen wie Kapselung und Vererbung beide Probleme adressiert, aber neben neuen Problemen keine endgültige Lösung bieten können. Eine der Ursachen hierfür ist, dass logisch zusammenhängender Code über mehrere Orte in der Programmierung verteilt ist. Dies führt zu schwer verständlichen und schwer wartbaren Programmen, da eine Veränderung einer Eigenschaft Anpassungen an mehreren Stellen erfordert. Das Prinzip *Separation of Concerns* wurde bei mehreren, unterschiedlichen Lösungsansätzen dieses Problemes in der Forschung verwendet, indem logische Eigenschaften durch eine Einheit (engl. *entity*) gekapselt wurden. Zwei dieser Lösungsansätze sind die Aspektorientierung und die generative Programmierung, die zwar unterschiedliche Perspektiven und Terminologien, aber auch viele Ähnlichkeiten haben.

AOP ist ein Meta-Programmiersprachen-Modell, das einzelne Aspekte, die mehrere Programmteile betreffen, konzeptionell gesehen an jeweils einer Stelle bündelt, d.h. in einem zunächst eigenständigen Codefragment darstellt. Eine Änderung eines Aspektes führt zu einem *Large Scale Refinement* ([Lieberherr, 1997]) der Anwendung, d.h. einer Änderung in mehreren Teilen des Programmes. Bei einer *generativen Programmierung* übersetzt ein Generator die domänenspezifische Beschreibung der Anwendung in Programmcode, der kompiliert werden kann. Eine Änderung eines Merkmales der gewünschten Anwendung kann auch hier zu einem *Large Scale Refinement* führen. Bei beiden Konzepten wird die Repräsentation der eigentlichen Systemfunktionalität gekapselt.

In der Aspektorientierung ist es möglich, bestehenden Code zu benutzen und durch die Kombination mit neuen Aspekten zu einer Vielzahl von Anwendungen zu gelangen. Sowohl Teile der Programmierung als auch die Aspekte können wiederverwedet werden. Durch eine Auswahl von Aspekten kann eine Vielzahl unterschiedlicher Anwendungen eines Problembereichs beschrieben werden. Im Vergleich dazu kann bei der generativen

Programmierung durch eine Architekturspezifikation in Verbindung mit der (Wieder-) Verwendung von komponierbaren Softwarebausteinen ebenfalls eine ganze Anwendungsfamilie generiert werden.

5.5.5 Aspekte vs. Komponenten

Die Komponentensicht fordert die Kapselung einzelner Programmteile und betont auf diese Weise Fragen der Integration und Kombination. Der Komponentenbegriff, wie er in der Aspektorientierung auftritt, kann mehr oder weniger direkt auf den Komponentenbegriff, wie er in Kapitel 2 umrissen wurde, abgebildet werden werden. Es stellt sich höchstens die Frage der Granularität. In einer reinen Komponentensicht ließen sich auch Aspekte zu einzelnen Systembausteinen in direkte Relation setzen, wenn gewissermassen eine Dienstleistungssicht eingenommen wird: Ein Aspekt wird durch eine Komponente funktional unterstützt und steht so den übrigen Systembestandteilen zur Verfügung. Bei einer vorangehenden Domänenanalyse sollten diese Aspekte identifiziert und auf Komponenten abgebildet werden, sodass die Trennung der Belange nicht auf Programmierebene vorgenommen werden muss. Eine ähnlich direkte Abbildungsmöglichkeit von Komponenten wurde schon bei der Umsetzung (»Verdinglichung«, reification) von Entwursmustern in [Griffel, 1998] aufgezeigt. Komponenten haben den Vorteil, dass sie auch nach einer Verknüpfung noch eigenständig sind.

5.5.6 Transformationssysteme

[Czarnecki, 1999] definiert eine Transformation als eine semantisch korrekte, automatisierte Modifikation eines Programms. Dabei wird eine Ersetzungsregel definiert, die aus einem Kopf- und einem Körperteil besteht. Wenn im zu übersetzenden Programm der Kopfteil gefunden wird, wird er durch den Körperteil ersetzt. Die Anwendung der Ersetzungsregel erfolgt dabei nicht direkt auf der textuellen Ebene, sondern im Syntaxbaum eines Programmes. Es wird ein Produkt der gleichen Art erzeugt, das auf der gleichen Abstraktionsebene liegt wie das Ursprungsprodukt. Einige Arten von Transformationen sind zum Beispiel Inlining, Vereinfachung, partielle Evaluierung, Refactoring und Verfeinerung.[30].

[30] vgl. zu deren praktischen Einsatz etwa `http://www.reasoning.com` oder `http://www.semdesigns.com`

Auch die derzeitigen Aktivitäten des World Wide Web Consortium (W3C) bieten mit der XSLT-Spezifikation[31] ein gutes Beispiel für einen Transformationsansatz. Die Extensible Stylesheet Language Transformation (XSLT) bietet ein Vokabular zur strukturellen Umformung eines XML-Dokumentes in ein anderes, d.h., erlaubt letztlich eine definierte Abbildung von benutzerdefinierten XML-Ontologien aufeinander. Damit lassen sich XML-Dokumente zum Beispiel in HTML-Dokumente umformen.

Der wichtigste Bestandteil von XSLT sind Schablonen (engl. *Template Rules*), die Anweisungen für einen XML-Parser enthalten, die dieser mit XML-Elementen aufüllt und kombiniert. Dazu wird mit Hilfe der XML Path Language (XPath)[32] der Pfad zu einem oder mehreren Knoten im Dokument beschrieben, wobei das Dokument als eine Baumstruktur betrachtet wird. Für jeden passenden Knoten im XML-Dokument wird die Template Rule ausgeführt. Die Schablonen enthalten enthalten Anweisungen für einen XSL-Parser, die dieser mit XML-Elementen auffüllt und kombiniert (vgl. das Navigieren im oben dargestellten Demeter-Ansatz).

5.5.7 Transformationssysteme vs. Generatoren

Transformationssysteme und Generatoren führen beide eine Umformung eines Entwicklungsartefakts in ein anderes durch, in der Regel wird ein Übergang zwischen zwei Programmrepräsentationen hergestellt. Im Gegensatz zu Transformationssystemen entsteht durch den Einsatz von Generatoren ein Produkt auf einer neuen Abstraktionsebene. Es findet mehr als eine einfache Umformung statt. Bei der generativen Programmierung übersetzt ein Generator eine abstrakte Anwendungsspezifikation in eine ausführbare Anwendung. Diese Ausgangsbeschreibung selber kann kaum als eigenständiges Programm betrachtet werden. Es können hierbei aber transformative Techniken angewendet werden, sodass meist ein fließender Übergang in der Begriffswahl auftritt. Generatoren wie etwa in 5.7.3 zur Erstellung von Matrizen[33] verwenden C++-Templates zur Erzeugung weiterer C++-Templates, sodass auch hier von einem Transformationssystem gesprochen werden kann.

[31] `http://www.w3c.org/TR/xslt`

[32] `http://www.w3c.org/TR/xpath`

[33] Programmkonstrukte, die die Verarbeitung des mathematischen Konzepts der Matrix erlauben.

5.5.8 Intentional Programming

Unter der Leitung von Charles Simonyi arbeitet Microsoft seit einigen Jahren an einem neuen Programmierparadigma [**?**]. *Intentional Programming*[34] liefert eine ganze Programmierumgebung und erlaubt es dem Programmierer durch die Einführung sogenannter Intentionen ein höheres Abstraktionsniveau in der Entwicklung zu betrachten. Bisher werden Progammabschnitte in der Regel nur durch Kommentarzeilen beschrieben. Mit Hilfe von Intentionen kann der Programierer seine Absichten explizit ausdrücken, wobei Intentionen in sehr abstrakter und nahezu beliebig problemspezifischer Form ausgedrückt werden können. Andere Programmierer müssen sich nicht mehr mit dem Quellcode auseinandersetzen, sondern erhalten anhand der Intention eine Beschreibung der Funktionalität des jeweiligen Programmteiles, der einfach für andere Programme wieder verwendet werden kann. Intentionen werden nach Bedarf von der Systemumgebung in herkömmliche Programmiersprachkonstrukte transformiert. Dem ganzen System liegt dabei ein datenbankbasiertes Transformationssystem zu Grunde.

Intentionen können standardisierte, domänenspezische Abstraktionen darstellen oder von den Programmierern selbst festgelegt werden. Dadurch dass der essentielle Inhalt, die Intention, sprachunabhängig repräsentiert wird, können unterschiedlichhe Programmiersprachen für die eigentliche Implementierung zum Einsatz kommen. Intentional Programming befindet sich in der Entwicklungsphase. Die Umsetzung in ein auslieferbares, kommerzielles Produkt wird noch einige Jahre benötigen.

5.6 Prinzip eines Software-Generators

In [Eisenecker et al., 1997] werden folgende Prinzipien der generativen Programmierung genannt:

- Variable Anteile werden identifiziert und als getrennte Parameter von abstrakten Komponenten modelliert (*Separation of Concerns* [Czar96])
- Abhängigkeiten und Wechselwirkungen werden identifiziert und explizit, möglichst an einer Stelle modelliert [Czarnecki, 1999]
- Komponenten werden für den jeweiligen Einsatzzweck optimal konfiguriert. Soweit als möglich, sind Kompo-

[34] `http://research.microsoft.com/ip`

nenten und Konfigurationen dafür selbst verantwortlich (*Intelligent Components* bzw. *Intelligent Configurations* [Batory, 1997][Czarnecki, 1999])

- Komponenten und Konfigurationen überwachen gegebenenfalls die Einhaltung bestimmter Anforderungen und Einschränkungen (*Configuration Rule Checking* und *Constraint Checking*)

- In der fertigen konkreten Komponente bzw. in einem fertigen Framework gibt es keine ungenutzte Funktionalität (*Zero Overhead Rule*)[35]

- aus der Redundanzfreiheit folgt die Fehlerminimierung

5.6.1 Generatoren als Compiler

Generatoren können wie schon erwähnt als Compiler für domänenspezifische Sprachen angesehen werden ([Smaragdakis & Batory, 2000]). Dies würde auch geschichtlich zu der Entwicklung der Programmiersprachen passen, die eine immer grössere Abstraktion von der eigentlichen Ausführung des Programmes im Rechnersystem bieten. Hier handelt es sich allerdings um domänenspezifische Sprachen, die nur für einzelne Domänen gelten und nicht den Anspruch einer allgemeingültigen Programmiersprache haben. Der Aufbau von Generatoren und Compilern ähnelt sich aber. Abbildung 5.3 zeigt den sehr ähnlichen Aufbau eines Generators mit einem Front-End, einer Übersetzungsmaschine und einem Back-End. Die Eingabe in der domänenspezifischen Sprache wird in eine vorübergehende Repräsentation übersetzt, genauso wie ein Compiler den Code in einen abstrakten Syntaxbaum übersetzt. In diesem ersten Schritt kann die Spezifikation auf semantische Richtigkeit überprüft werden. Generatoren wie Jakarta (s. Abschnitt 5.7.2), die ihrerseits wieder Generatoren generieren,

[35] Dieser Regel unterwirft sich der hier vorgestellte Ansatz nicht strikt, da einzelne Segmente einer Komponente (vgl. a. Kapitel 2) ungenutzt bleiben können. Es steht hier nicht die optimale Kompaktheit des Anwendungscodes im Vordergrund. Auf Grund der klaren Lokation der Einzelfunktionalitäten im Komponentenansatz ist aber beispielsweise ein abschließender Optimierungsschritt für erstellte Anwendungen denkbar, der ungenutzte Segmente einer Komponente entfernt (vgl. a. Techniken der Refaktorisierung [Fowler et al., 2000]). Entsprechende Möglichkeiten wurden im Rahmen der Arbeit aber nicht vorgesehen.

die letztendlich domänenspezifische Erweiterungen einer Programmiersprache (in diesem Falle Java) erlauben, verdeutlichen die Ähnlichkeit von Generatoren und Compilern.

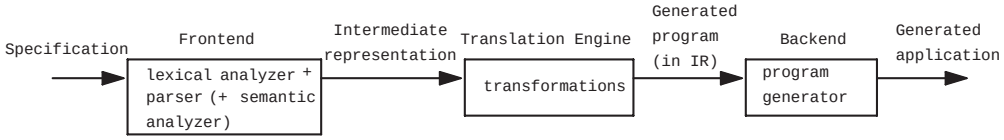


Abbildung 5.3
Aufbau eines Generators in Analogie zu einem Compiler

5.6.2 Frameworks und Generatoren

Rahmenwerke (engl. *Frameworks*) stellen eine gängige, allerdings meist recht technisch orientierte Art der Repräsentation von Domänenwissen dar und erlauben es, ein vorgegebenes Grundgerüst an spezielle Einsatzfälle und Bedürfnisse anzupassen bzw. entsprechend zu erweitern. In der Regel werden abstrakte Klassen dabei konkret implementiert oder vorgefertigte Klassen durch Vererbung für die eigenen Anforderungen des Entwicklers spezialisiert. Das Domänenwissen ist in diesem Fall durch die einzelnen Funktionalitäten der Klassen bzw. deren gegenseitige Abhängigkeitsstruktur impliziert. Es wird nicht explizit ausgedrückt wie bei einem generativen Ansatz mit Hilfe domänenspezifischer Sprachen. Jede Anpassung an die eigenen Bedürfnisse erfolgt im Programmcode, sodass Anpassungen und Detailveränderungen häufig komplexe Abhängigkeiten berücksichtigen müssen. Hilfsmittel wie ein »Refactoring« versuchen heute solche Probleme zu adressieren [Fowler et al., 1999], stellen in der Sicht des Autors aber wiederum ein »Nachbessern« statt ein *konstruktives* Verfahren dar.

[Czarnecki, 1996] sieht allerdings bei einer Weiterentwicklung von Frameworks eine Annäherung an domänenspezifische Sprachen. Der erste Entwurf eines Frameworks ist meist ein Whitebox-Framework. Erst durch Erfahrungen, die bei der Verwendung des Framework ansammeln, entsteht ein verbessertes Framework, das anstelle der ursprünglichen Vererbungsmechanismen auf anderen, weniger problematischen Anpassungsmechanismen wie etwa einer Parametrisierung aufbaut.[36] Das resultierende Blackbox-Framework hat dann einen stark problembezogenen Charakter. Der Code hat nach der Vorstellung von [Czarnecki, 1996] dann Ähnlichkeiten mit einer domänenspezi-

[36] Zur Framework-Diskussion vgl. a. [Griffel, 1998]

fischen Sprache.[37] Die Überführung der in einer domänenspezifischen Sprache geschriebenen Programme, in solche, die auf einem Framework basieren, kann wiederum mittels eines Generators erfolgen.

Beim Entwurf einer domänenspezifischen Sprache sollten die Prinzipien der Aspektorientierung zum Einsatz kommen. Es zeigt sich, dass Rahmenwerke, aspektorientierte Programmierung und Generatoren komplementäre, sich gegenseitig befruchtende Ansätze darstellen.

5.6.3 Entwicklungsprozesse und Generatoren

Die generative Programmierung erlaubt zum Zeitpunkt der eigentlichen Systemerstellung keine partizipative Beteiligung des Endkunden[38], wie dies bei modernen Entwicklungsprozessen der Fall sein sollte [Griffel, 1998]. Die Domäne muss vor der Auslieferung des Produktes fertig analysiert worden sein und genau hier kann ein Kunde eng eingebunden werden, insbesondere da nur auf einem fachnahen Niveau diskutiert werden muss. Es besteht nicht direkt die Möglichkeit, iterativ die Entwicklung zu begleiten. Allerdings entsteht bei der Domänenanalyse ein Modell, das alle möglichen Anwendungen oder alle Varianten eines Anwendung innerhalb eines Anwendungsbereichs umfasst. Wenn der Endkunde mit der ersten Version der Anwendung nicht zufrieden ist, kann eine andere Version generiert werden. Der Generator erlaubt es, mehrere unterschiedliche Versionen der Anwendung rasch und kostenminimiert zu erstellen. Auf diese Weise ist es möglich, die Wünsche der Endkunden bei der Erstellung der Anwendung zu berücksichtigen.

5.7 Beispiele generativer Ansätze

Im Folgenden sollen kurz einige konkrete Beispiele existierender generativer Ansätze betrachtet werden, um unterschiedliche Merkmale und gemeinsame Schwerpunkte zu verdeutlichen.

5.7.1 GenVoca

GenVoca ist aus zwei unterschiedlichen domänenspezifischen Softwaresystemgeneratoren, Genesis und Avoca, hervorgegan-

[37] Diese Ansätze sind allerdings derzeit rein theoretisch und wurden bislang nicht durch die Entwicklungspraxis verifiziert.

[38] Allerdings könnte ein Kunde bei der Merkmalsauswahl involviert werden (vgl. a. Abschnitt 5.11.1)

gen. In [Batory, 1998] wird deutlich, wie Generatoren benutzt werden können, um eine Produktlinienarchitektur (s. Abschnitt 5.2.1) zu implementieren. [Batory & Geraci, 1997] nennt GenVoca ein domänenunabhängiges Modell für die Definition von skalierbaren Familien von hierachischen Systemen aus Komponenten. GenVoca bietet ein einfaches Modell mit klarer Notation, um die Systeme, die durch Generatoren produziert werden sollen, zu repräsentieren. Das Modell ist möglichst generell gehalten, um eine einheitliche Methode zur Domänenanalyse zu unterstützen. Das Modell wurde schon in mehreren unterschiedlichen Domänen, wie Datenstrukturen, Datenbanken, Dateisystemen, Netzwerkprotokolen und in der Luftfahrt angewendet.

Die GenVoca-Architektur ist eine Schichtenarchitektur, die eine geschichtete Aufteilung der Implementation verlangt. Eine Schicht (*layer*) kann aus mehreren Basiskategorien (*realms*) bestehen. Eine Schicht kapselt dabei ein primitives Merkmal der Domäne. Die Implementierungen einzelner Schichten können ausgetauscht werden. Eine Basiskategorie kann von unterschiedlichen Komponenten implementiert werden, die alle nach aussen die gleiche Schnittstelle exportieren, aber unterschiedliche Schnittstellen importieren können, sodass sich auf natürliche Weise eine Hierarchie ergibt. Komponenten können in dieser Sicht aus Quellcode bestehen, der zur Laufzeit der Anwendung ausgeführt wird oder aber Meta-Programme sein, die den benötigten Quellcode erst generieren.

Die Kategorien stellen eine fundamentale Domänenabstraktion dar. Durch eine relativ kleine Anzahl von Komponenten kann eine große Anzahl von Kombinationen abgeleitet werden. Eine GenVoca-Architektur besteht im wesentlichen aus 3 Bestandteilen:

- eine formale Architekturbeschreibung, in der die Basiskategorien der Architektur und ihre Beziehungen untereinander genannt werden,
- eine formale Beschreibung der Schnittstellen dieser Basiskategorien,
- eine komplette Spezifikation für jede Komponente jeder Basiskategorie, wie die Komponente die Schnittstelle der Kategorie erfüllt.

Die Spezifikation einer konkreten Anwendung erfolgt dann durch eine Gleichung (*type equation*) mit den gewünschten Komponenten als Parameter, wobei innere Komponenten durch äussere verfeinert werden. Beim diesem *Refinement* wird eine

abstrakte Schnittstelle in eine konkrete Implementation umgewandelt.[39] Im Folgenden wird ein einfaches Beispiel für die Definition von Basiskategorien und für die Spezifikation einer Anwendung durch eine entsprechende Gleichung gezeigt.

Basiskategorien:
`R = y, z, w`
`S = g[ x:R ], h[ x:R ], i[ x:R ]`
Definition einer Anwendung:
`Anw = g[y] oder Anw = h[w]`
Erläuterung:
Für das Realm R existieren 3 Komponenten, die die Basiskategorie implementieren. Für das Realm S exisitieren ebenfalls 3 Komponenten, wobei Komponenten des Realm R als Parameter für die Spezifikation übergeben werden. Die Spezifikation der Anwendung erfolgt durch Gleichungen wie `g[y]` oder `h[w]`. Alle Komponenten von Realm S sind in diesem Beispiel Transformationen von Realm S zum Realm R ([Solderitsch et al., 1995]). Die Parameter geben dabei die Schnittstellen an, die die Komponenten benötigen, wobei die Parameter für die einzelnen Komponenten eines Realms unterschiedlich ausfallen können. Alle Komponenten, die R implementieren, können die Bedingungen der Komponenten g, h und i erfüllen, die von Realm R abhängen. *Symmetrische* Komponenten ex- und importieren die gleiche Schnittstelle (vgl. a. Kapitel 2). Auf diese Weise können mehrere Komponenten einer Kategorie an einer Stelle eingesetzt werden. Die Komponenten einer Basiskategorie können, müssen aber nicht semantisch zueinander kompatibel sein.

Gleichungen wie `Anw = g[y]` geben alle möglichen Anwendungen einer Produktlinie an. Dabei können einige Kombinationen von Komponenten zwar syntaktisch korrekt sein, aber semantisch keinen Sinn machen. Hierfür müssen *Konfigurationsregeln* definiert werden. Bei der Übersetzung der Auswahlparameter kann ein Generator eine domänenspezifische Optimierung durchführen, indem domänenspezifische Entscheidungen bei der Wahl bestimmter Algorithmen getroffen werden. Die Auswahl eines Parameters kann zu einer Änderung in mehreren Klassen der Endanwendung führen. Es wird in diesem Fall ein *Large Scale Refinement* vorgenommen. Das Modell selbst besagt nicht, wie diese Verfeinerungen implementiert werden. Die eigentli-

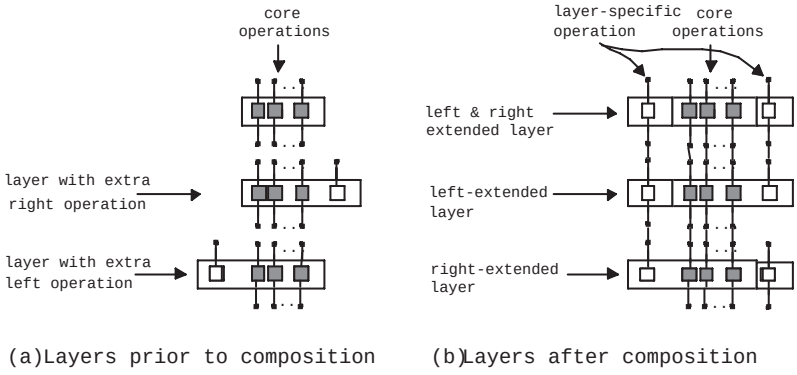
[39] Refinement ist in diesem Sinne also als Übergang *zwischen* zwei Abstraktionsebenen zu interpretieren, während es ansonsten oftmals nur eine Verfeinerung *innerhalb* eines Abstraktionsniveaus kennzeichnet.

chen Bausteine der erzeugten Anwendung können aus der Transformation der Verfeinerungen in konkrete Komponenten entstehen. Es ist aber auch möglich, durch kompositorische Verknüpfung von vorhandenen Komponenten eine Anwendung zu erzeugen.[40]

5.7.1.1 Subjektive Schnittstellen

GenVoca beruht auf der Verwendung von Standardschnittstellen und geht dabei davon aus, dass Schnittstellen eine Kernschnittstelle (engl. *core interface*) besitzen, die durch optionale Operationen (zum Beispiel Methoden) erweitert werden kann (vgl. a. Kapitel 2). Ein `Textbuch` stellt etwa ein Beispiel für die Notwendigkeit von subjektiven, d.h. situationsbedingten Schnittstellen dar, da nicht alle Anwendungen einer Domäne die gleiche Schnittstelle exportieren: In einer Buchhandlung sind Angaben über den Autor, den Titel und den Inhalt hilfreich. In einem Lager wird nur die ISBN-Nummer oder eine intern verwendete Nummer benötigt, um das Buch im Regal zu finden, sowie die Anzahl der vorhanden Exemplare eines Buches (vgl. a. Abschnitt 4.8).

Abbildung 5.4
Subjektive Schnittstellen bei GenVoca



Die Schnittstelle, die letztendlich exportiert wird, hängt von den verwendeten Komponenten ab. Abbildung 5.4 zeigt, wie eine Komponente, die eine zusätzliche Methode ausserhalb der

[40] Dies sind allerdings theoretische Postulate des GenVoca-Ansatzes, da der Kompontenbegriff softwaretechnisch keine exakte Entsprechung besitzt ist und ein definiertes Verfahren zur Überführung entsprechender Gleichungen in konkreten Code (Anwendungsskelett) fehlt. Die guten Praxisresultate GenVocas basieren auf speziellen, jeweils Projektspezifischen Definitionen.

Kernschnittstelle anbietet, zu einer Veränderung aller verwendeten Komponenten führt, sodass diese Methode auch nach aussen exportiert wird. Die optionalen Operationen werden bei der Komposition für alle anderen Schichten hinzu generiert. Wenn die Schicht mit der optionalen Operation aus der Spezifikation der Anwendung entfernt wird, wird auch die optionale Operation nicht hinzugefügt. Auf diese Weise werden Familien von Schnittstellen generiert, die auf gleichen Kernschnittstellen beruhen. Dies entspricht einer *subjektorientierten Programmierung* [Harrison, 2000][Blando, 2000]).

Nach [Batory & Geraci,] drückt Subjektivität die Variabilität von Objektschnittstellen aus. Interessant ist also, wo die Subjektivität angesiedelt ist. Denkbar sind etwa Komponenten mit Einheitsschnittstellen im Sinne etwa der Sun Studio-Beans [Griffel, 1998], die dann einfach zu komponieren sind, deren Intelligenz / Subjektivität aber durch die interne Auswertung / Verarbeitung der eingehenden Daten realisiert ist. Das Problem solcher »intelligenter« Komponente ist, dass eventuell zuviel nicht-problembezogene Zusatzfunktionalität in sie implementiert werden muss, die sich um die Auswertung des Datenflusses kümmert, aber eben nicht zur Anwendungslogik beiträgt. Eine Flussregelung durch intelligente, aber externe Konnektoren wie sie Kapitel 3 als Adapter vorschlägt präferiert der Autor daher.

5.7.1.2 Validierung von Kompositionen

Es wurde schon betont, dass nicht alle syntaktisch korrekten Gleichungen auch semantisch korrekte Anwendungen ergeben. GenVoca hat einen einfachen auf Name-Werte-Paaren basierenden Mechanismus, der dafür sorgen soll, dass die ausgewählten Komponenten letztendlich auch zusammenpassen. Es wird dabei zwischen im- und exportierten Eigenschaften unterschieden. Eigenschaften, die eine Komponente an eine »untere« Komponente[41] exportiert, sind normalerweise auch nicht die gleichen, die sie an eine »obere« weitergibt. Auch die Richtung spielt also eine Rolle. Bei der Komposition wird überprüft, ob die Funktionsspezifikation gültig ist. Es wird dabei zwischen Pre- und Postconditions und zwischen Pre- und Postrestrictions unterschieden. Eine Postcondition ist eine Eigenschaft, die eine Komponente nach

[41] Die Ausdrücke »unten« und »oben« beziehen sich auf die relative Position in der Gleichung. Die äußerste Komponente in der Gleichung ist die »höchste« Komponente und die innerste wird als die »niedrigste« bezeichnet.

»unten« exportiert. Bei einer Precondition muss eine Komponente »über« ihr diese Eigenschaft als Postcondition spezifiziiert haben, damit die Gleichung gültig ist. Bei einer Prerestriction muss die Eigenschaft als Postrestriction einer in der Hierarchie »unter« ihr liegenden Komponente gegeben sein.

5.7.2 Jakarta

Nach [Batory et al., 1997] fehlt die Unterstützung in Programmiersprachen und Werkzeugen für die Erstellung von Softwaregeneratoren. 60% der notwendigen Anstrengungen für einen domänenspezifischen Generator betreffen die domänen*unspezifische* Programminfrastruktur, wie Sprachen für die Komponentenspezifikation und -komposition und Ähnlichem. Jakarta soll diese Infrasturktur bereitstellen und liefert mit der Jakarta Tool Suite (JTS) ein Werkzeug für die Erstellung von Generatoren.

Es wird eine Art Compiler für domänenspezifische Sprachen erstellt, indem nicht erweiterbare Programmiersprachen wie etwa Java derart flexibilisiert werden, dass domänenspezifische Erweiterungen möglich sind. Java wurde dabei zum Beispiel um Konzepte zur Metaprogrammierung wie *backquotes* und *comma* aus LISP erweitert. [Batory et al., 1997] sieht einen Generator entweder als Compiler einer domänenspezifischen Sprache oder aber als Compiler für eine *general-purpose*-Sprache, die um genau die domänenspezifischen Elemente erweitert wurde, die für die jeweilige Entwicklung gerade benötigt werden. Jakarta verbindet somit domänenspezifische Sprachen / Compiler-Technologie mit Softwarekomponenten / Generator-Ansätzen. Die JTS ist dabei selbst in einer eigenen Erweiterung von Java geschrieben. Es wird dann die GenVoca-Methodik angewendet, um einen Compiler für domänenspezifische Sprachen zu generieren.

5.7.3 C++ Templates

Auch [Eisenecker & Czarnecki, 1999] propagiert eine generative Programmierung. Der Ansatz beruht hier auf der Benutzung von C++-Templates zur generativen Erstellung *von* Komponenten. Hierbei erfolgt die Spezifikation des Systems, genaugenommen einer Komponente, durch die Angabe der gewünschten Merkmale und Eigenschaften. Durch Transformationen wiederum mit Hilfe von C++-Templates werden dabei die eigentlichen Zielkomponenten generiert. [Eisenecker & Czarnecki, 1999] verwendet hierbei

für C++-Templates den Begriff der Komponente synonym, vertritt also eine »klassische« Komponentendefinition auf Quellcodeebene.

5.7.3.1 Das Template-Konzept

C++-Templates bilden durch die Bereitstellung von *parametrisierten Typen* die Grundlage für eine *generische Programmierung* (vgl. [Cheong & Jarzabek, 1999], [Musser & Stepanov, 1994] und [Musser & Stepanov, 1989]). Dabei existieren Klassen-Templates, die Familien von Klassen kennzeichnen, und Funktionen-Templates, die Familien von Funktionen bereitstellen. Die Templates lassen den Austausch von Quellcode zu, sodass eine Verfeinerung ohne Vererbung möglich ist.

5.7.3.2 Vorgehensweise

Es werden zunächst, wie in Abschnitt 5.3 dargestellt, in einer Domänenanalyse Gemeinsamkeiten, Unterschiede und Wechselwirkungen der Merkmale der Domäne analysiert. Das Ziel der Domänenanalyse ist in diesem Fall die Entwicklung einer fachspezifischen Sprache zur Beschreibung der gewünschten Eigenschaften einer Anwendung. Diese Sprache zur Beschreibung des Problemraumes wird als Domain Specific Language (DSL) bezeichnet. Die DSL hat als Repräsentation parametrisierte C++-Templates. Die entwickelte Sprache erfordert keine Kenntnis über Implementierungsdetails. Die Eigenschaften der Domäne können übersichtlich in Merkmalsdiagrammen dargestellt werden, wobei die Wechselwirkungen der Eigenschaften, d.h. unmögliche Kombinationen von Merkmalen, separat in Textform oder tabellarisch dargestellt werden müssen. Es ist ebenfalls wichtig Standardvorgaben für Eigenschaften, die vom Anwender nicht spezifiziert wurden, zu bestimmen.

Als nächstes muss beim Domänendesign eine gemeinsame Archiktektur für Systeme einer Domäne entwickelt werden. Es muss modelliert werden, welche Komponenten zu implementieren sind. Das Vorbild für die Systemarchitektur liefert wiederum das oben vorgestellte GenVoca-Modell.

Bei der Domänenimplementierung wird eine (Teil-) Menge des Domänenmodells implementiert. Zur Beschreibung des Lösungsraumes existiert eine eigene Sprache, die Implementation Components Connection Language (ICCL). Sie beschreibt, welche elementaren Bausteine auf der untersten Ebene existieren und wie sie kombiniert werden können. Diese implementierungs-

spezifische Sprache beschreibt eine Konfiguration von Implementierungskomponenten. Die Aufgabe eines Generators ist es dann, die Beschreibung des Problemraumes in der DSL in eine Beschreibung des Lösungsraumes in der ICCL umzuwandeln. Bei einer Änderung der DSL muss die Übersetzung in die ICCL entsprechend angepasst werden. Auch die ICCL ist letztendlich als C++-Template realisiert.

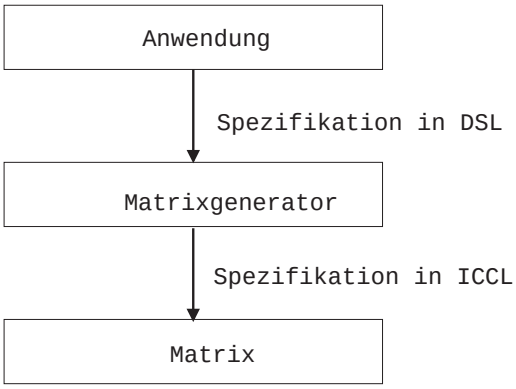


Abbildung 5.5 *Schematischer Aufbau eines Generators zur Matrix-Erstellung*

5.7.3.3 Beispielanwendung von C++-Templates

In [Neubert, 1998] wird das geschilderte Verfahren etwa zur Erstellung von Matrizen verwendet. Das folgende Beispiel zeigt eine Beschreibung einer Matrix in der DSL und deren Übersetzung in die entsprechende ICCL :

DSL:
matrix<int, structure<>, space<>, check_as_default<>, check_bounds<> >
ICCL:
Matrix <RectMatrix <BoundsChecker <Dyn2DCContainer <Config> > > >

Eine Matrix wird durch die Angabe mehrerer Parameter konfiguriert. Der erste Teil der DSL beschreibt den Typ der Elemente der Matrix, in diesem Fall `int`. Die Struktur der Matrix (structure) ist nicht näher spezifiziert, sodass Standardvorgaben genutzt werden. Die Beschreibung der Struktur reicht von der Anordnung der Elemente über die zu verwendende Datenstruktur bis hin zur Methode der Speicherbeschaffung. Es wird nach Speicherbedarf (`space`), nicht nach Ausführungsgschwindigkeit optimiert. Es wird eine Fehlerbehandlung vorgenommen

(`check_as_default`) und die Index-Bereichsgrenzen werden erzwungen(`check_bounds`).

Die ICCL beschreibt die Komponenten, die für die Implementierung der spezifizierten Matrix eingesetzt werden. Es wird eine rechteckige, allgemeine Matrix verwendet, die die Komponente BoundsChecker verwendet. Die Komponente Dyn2DCContainer liefert die Funktionalität der Datenstruktur. Config wird während der Generierung erzeugt und beschreibt die generierte Matrix, wodurch alle wesentlichen Eigenschaften in jeder Schicht bekannt sind.

Die Übersetzung der DSL in die ICCL erfolgt ebenfalls durch die Anwendung von C++-Templates, sodass für den Anwender die Benutzung des Generators transparent erfolgt. Es wird ein neues Template durch die Evaluation eines bestehenden Templates erstellt.

5.7.3.4 Bewertung

Die Verwendung von C++-Templates ermöglicht die Erstellung einer Vielzahl unterschiedlicher, optimal konfigurierter Komponenten auf Quellcode-Ebene. Die Betonung liegt hierbei auf der Entwicklung von Komponenten. Es wird nicht deutlich, wie weit sich diese Vorgehensweise eignet, um grössere Systeme zu generieren. Die generative Programmierung beruht hier auf einer Ausreizung der C++-Template-Technologie. Der »Generator»ist dabei letztlich der C++-Compiler selbst und der ganze Ansatz damit sehr auf die (spezielle) Sprachebene ausgerichtet.

5.7.4 Convergent Architecture

Zur Vervollständigung der bislang betrachteteten Ansätze aus dem akademischen Umfeld soll hier noch kurz ein kommerzielles System angesprochen werden, da es dem in dieser Arbeit propagiertem Vorgehen am nächsten kommt.

Die Firma Interactive Objects (IO)[42] hat eine auf architektonischen Stil spezialisierte Umsetzung der besten Aspekte aus den modernen Entwicklungsmethoden *OPEN*, *Rational Unified Process* [Jacobson et al., 1999] und *Catalysis* [D'Souza & Wills, 1998b] entwickelt [Huber & Elting, 1999]. Der zu Grunde liegende Prozess beruht auf dem *Convergent Engineering*, das 1995 von David A. Taylor eingeführt wurde. Es sich um eine Designvision, die Geschäftsmodellierung und die IT-Systementwicklung vereint. Dabei wird mit Hilfe fundierter

[42] `http://www.io-software.com`

Kenntnisse der Objektorientierung ein Unternehmen bzw. dessen Abläufe modelliert. Dieses Modell wird möglichst bruchlos in Software abgebildet. Auf diese Weise entsteht ein Modell, das von Geschäftsleuten und von der IT-Abteilung wiederum gleichermassen verstanden und getragen wird. Veränderungen im Unternehmen können im IT-Modell modifiziert werden, ohne auf die technologische Ebene zu achten.

Der gesamte Lebenszyklus eines IT-Systems wird durch den Einsatz von Werkzeugen unterstützt. Für die Implementierung der Modelle werden Enterprise JavaBeans und CORBA Components eingesetzt. Aspekte wie automatische Persistenz, Transaktionsicherung, Zugriffsschutz und Metainformationen ähneln sich in allen Komponenten, hängen aber in Details von der Art der Komponente ab. Daher wird dieser Teil des Codes automatisch generiert und dadurch gleichzeitig eine gleichbleibend definierte Qualität des Komponentenquellcodes erreicht. Komponenten enthalten dabei Metainformationen im CDF-Format, einer Auszeichnungssprache analog zu XML, sodass alle Informationen über die Komponente zentral bei der Komponente verfügbar sind.

Ähnlich wie bei obigem Template-Ansatz werden hier auf Quellcodeebene Komponenten generiert. Bei der Convergent Architecture handelt es sich aber um einen ganzheitlichen Ansatz, der auch die Verwendung der Komponenten in einer Anwendung mit einschließt. Interessant ist hierbei auch die Verwendung von Komponentenschnittstellen, um Metainformationen über Komponenten zu speichern.

Wiederum zeigen sich die bei allen betrachteten Ansätzen zu beobachtenden Ziele und ähnlichen Herangehensweisen zu deren Erreichung: Jeweils bildet die automatische Auswertung und Verarbeitung abstrakt gehaltener, fachnaher Informationen über getrennt betrachtete Systembestandteile die Grundlage für die tatsächliche Realisierung des gewünschten Zielsystems. Im weiteren wird daher die Konzeption des eigenen Ansatzes dieser Arbeit unter genau diesem Leitbild dargestellt.

5.8 Konzeption eines Deskriptor-gesteuerten Generators

5.8.1 Einordnung

Es gibt unterschiedliche Wege, wie ein Generator Programmcode aus einer Spezifikation konstruieren kann. [Batory, 1997] nennt

zum einen Unterschiede, wie und wann eine Verfeinerung der erzeugten Anwendung erfolgt. Wenn eine Anwendung aus vorhandenen Bausteinen zusammengestellt wird, spricht er von einem kompositorischen Ansatz (engl. *compositional approach*). Eine andere Möglichkeit besteht in der Verwendung von Generatoren, die die eigentlichen Bausteine der Anwendung erst generieren und diese dann zusammenfügen. Der letztere Ansatz wird dann als transformiernder Ansatz bezeichnet.

Eine weitere Unterscheidung betrifft den Zeitpunkt, zu dem die Bausteine zusammengefügt werden. Eine Verfeinerung der Anwendung kann zur Laufzeit erfolgen (*dynamisch*) oder zu einem vorherigen Zeitpunkt (*statisch*). Dynamische Ansätze bieten hierbei die grössere Flexibilität durch mögliches *late binding* von Komponenten. Statische Ansätze ermöglichen eine verbesserte Optimierungsmöglichkeit und damit eine bessere Performanz, dafür aber keine dynamische Rekonfiguration.[43]

Die im vorangegangenen Abschnitt vorgestellten Generatoren sind in erster Linie für die Erstellung von einzelnen Komponenten im Sinne dieser Arbeit vorgesehen. Die Implementierung ganzer Anwendungen steht nur im Convergent Engineering im Vordergund, da es einen ganzheitlichen Ansatz zur Erstellung von Applikationen aufweist, wobei allerdings wiederum einzelne Komponenten für die Gesamtanwendung spezifisch generiert werden.

Alle Ansätze arbeiten auf Quellcodeebene. Häufig wird zwar der Begriff der Komponente verwendet, wobei der Begriff aber gemäss der zu allgemeinen Definition von D'Souza verwendet wird, der u.a. auch einzelne Methoden und Sourcecode miteinschließt (s. Abschnitt 1.1.6). Der Ansatz dieser Arbeit liegt dagegen gerade in der Erstellung eines Applikationsgenerators, der von der Quellcode-Ebene weitestgehend abstrahiert und *vorgefertigte* Standardbausteine verwendet, die durch eine Rahmenanwendung lediglich miteinander verknüpft werden. Eventuell generierter Quellcode dient ausschließlich der Verknüpfung und Konfiguration von fertigen Komponenten und bleibt dem Softwarekonstrukteur verborgen.

[43] Fragen der Optmierung stehen nicht im Fokus dieser Arbeit, da Komponenten unverändert verwendet werden sollen und nicht deren Code optimiert werden soll. Dies stellt einen Kompromiss dar, da sonst niemals ein Komponentenkonzept auf implementatorischer Ebene (Codeebene) sondern immer nur im Model umgesetzt werden kann. Eine Optimierung ist daher nur bei der Verknüpfung (engl. *Gluing*) bzw. durch das Auslassen oder Umordnen von Komponenten möglich.

Generative Ansätze wie DiSTiL [Smaragdakis, 1997] oder der Template-Ansatz von [Czarnecki & Eisenecker, 2000], betonen den statischen und transformierenden Ansatz eines Generatoren. In dieser Arbeit soll ein Generator konzipiert werden, der ebenfalls statisch ist, jedoch eine Anwendung durch reine Komposition von Komponenten zusammenstellt. Es wird somit ein statischer und kompositorischer Ansatz verfolgt.

Im Folgenden werden die grundlegenden Konzepte des Generators zur Unterstützung einer angeleiteten, generativen Softwarekonstruktion beschrieben. Es wird gezeigt, wie mit Hilfe von typisierten, adaptierbaren Komponenten gemäß den Ergebnissen der vorangegangen Kapitel sowie einer Beschreibung von möglichen bzw. gewünschten Anwendungsstrukturen Applikationen weitgehend automatisch generiert werden können. Exemplarisch wird mittels der Anwendung »Taschenrechner« gezeigt, wie eine Domäne für das Generatorkonzept aufgearbeitet werden muss und wie die entsprechenden Modelle direkt und bruchlos in den generativen Vorgang einfließen.

5.8.2 Motivation

Die Erzeugung einer Anwendung beginnt bei den vorgestellten Ansätzen auf der Quellcode- und damit einer sehr niedrigen, problemfernen Ebene. Die Erzeugung von größeren Anwendungen würde um ein vielfaches erleichtert werden, wenn sie auf Grundlage vorhandener oder zumindest teilweise vorhandener Komponenten beruhen würde. Das Ideal wäre ein wachsender Markt von vorgefertigten Komponenten. Generatoren sollten dieses Reservoir an vorgefertigten Bauteilen nutzen können. Es nicht möglich sein, in allen Fällen vorgefertigte Komponenten zu finden, die genau den jeweiligen Bedürfnissen entsprechen, sodass eventuell eine Anpassung notwendig wird. Obwohl insbesondere für Fragestellungen von Integrationstests und der Validierung bei der Komposition von Komponenten noch intensive Forschung notwendig ist, erlaubt bereits der jetzige Kenntnisstand eine interessante und gewinnbringende Kombination generativer und komponentenbasierter Ansätze für eine architektur- bzw. modellgetriebene Anwendungskonstruktion.

Komponenten werden vom Konzept her für die Komposition mit anderen Bausteinen erstellt. In den vorangegangenen Kapitel wurden erweiterte Beschreibungen für Softwarekomponenten eingefürht, die nähere Angaben über die Art ihrer Verwendung und ihrer Bedürfnisse im Rahmen einer Produktarchitektur auszudrücken vermögen. Diese Angaben bieten eine ausgezeichnete

Grundlage, um von einem Generator ausgewertet zu werden. Auf dieser Basis ist es erheblich einfacher, eine Anwendung durch eine automatische Komposition von Komponenten zusammenzustellen, als dies mit herkömmlichen »low-level « Programmkonstrukten möglich ist, da eine klarere Abgrenzung und deutliche Rollenverteilung von Einzelteilen mit genügend hohem Funktionsumfang vorliegt. Der Generator muss allerdings Sorge tragen, dass die verwendeten Komponenten in die gewünschte Anwendungsstruktur »passen« und auch untereinander kompatibel sind. Hierfür steht ein vollständiges Typmanagment bereit, das wiederum auf der Auswertung von Komponenten-Deskriptoren basiert. Dies erlaubt es insbesondere auch, sehr komplexe Anwendungen ausgehend von überschaubaren Spezifikationen zu erstellen.

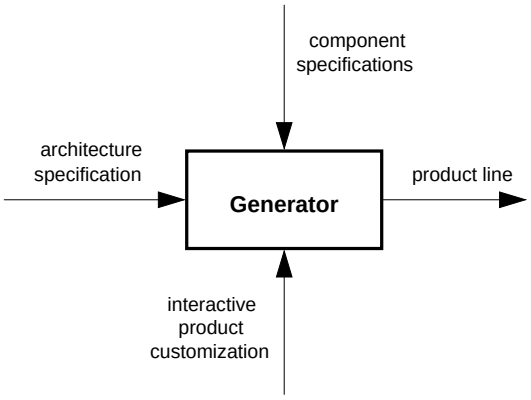


Abbildung 5.6
Der Generator erhält verschiedene Eingaben und kann Applikationen einer ganzen Produktlinie liefern.

Die Mehrheit der exisitierenden Ansätze ist nur für jeweils eine einzelne Domäne einsetzbar. So sind etwa die vorgestellten C++-Templates nur für die Domäne der Matrizen geeignet. Für eine neue Domäne müssten entsprechend neue Templates formuliert werden. Das GenVoca-Modell wurde aber durchaus für unterschiedliche Generatoren zur Unterstützung unterschiedlicher Domänen eingesetzt. Bei *Jakarta* werden spezielle Generatoren für jeweils eine Domäne erst ihrerseits generiert. Ein Ziel des Autors liegt in der Entwicklung eines Generators, der in mehreren Domänen eingesetzt werden kann. Dafür muss es möglich sein, die benötigten, domänenspezifischen Informationen einfach austauschen bzw. durch Parametrisierung zuführen zu können. Ausgangsbasis hierfür bildet zunächst die Wahl einer generischen Verarbeitungsmöglichkeit, um mit Hilfe eines stets gleichen Ver-

arbeitungsmechanimus die Merkmale und Termini unterschiedlicher »Domänensprachen« auswerten zu können.

5.8.3 Grundidee

Es soll also gewissermassen ein »Meta-Generator« konzipiert werden, der es auf einfache Weise erlaubt, eine Vielfalt von individuellen Anwendungen einer jeweils vorgegebenen Domäne (bzw. für sie gültigen Systemarchitekturen) aus einzelnen Komponenten zu konstruieren. Die generierte Anwendung besteht dann aus einer Komposition von lose gekoppelten Softwarekomponenten, also nicht direkt »verwobenem« Code. Dies bedeutet zwar einen Verzicht auf Optimierungsmöglichkeiten in der Programmierebene, sieht diesen Kompromiss jedoch bewusst vor, da der Autor die robuste und rasche Erstellung komplexer Systeme als vorrangiges Ziel der heutigen Softwareentwicklungs betrachtet, die es sich allgemein nicht mehr leisten kann, sich mit dem besten »Hack« pro Zeile aufzuhalten. Der Generator erhält die für die Generierung notwendigen Informationen zur Erstellung von korrekten Anwendungen aus Beschreibungen der Anwendungsarchitektur und der einzusetzenden Komponenten selbst. Es ist dem Anwender möglich, individuelle Anwendungen zu erhalten, indem er die Merkmale der gewünschten Anwendung auswählt. Dabei wird er durch einen graphischen Assistenten unterstützt und angeleitet, sodass beispielsweise für eine Domäne unzulässige Merkmals-Kombinationen ausgeschlossen werden können (vgl. a. Abschnitt 5.7.1.2). Eine Anpassung (bzw. Verfeinerung) der Anwendung kann in einer Post-Konfigurationsphase vorgenommen werden. Der Generator ist dabei so flexibel zu halten, dass er auch diese Konfigurationsmöglichkeiten unabhängig von (nur) einer spezifischen Domäne bereitstellen kann.

Die Beschreibung der gewünschten Anwendungsarchitektur für ein domänenspezifisches Problem erfolgt in Form einer Document Type Definition (DTD). Eine DTD beschreibt eigentlich gültige Dokumente eines Dokumententyps für XML-Dokumente. Die Grammatik der DTD ist aber auch mächtig genug, um mögliche Strukturen von Anwendungen beschreiben zu können, gerade wenn diese wie für die Komponentensicht gefordert strenge Hierarchien (azyklisch, linear und unter Minimierung der gegenseitigen Abhängigkeiten – vgl. a. das »Law-of-demeter« in Abschnitt 5.5.2 bzw. die Diskussion in Kapitel 2) bilden sollten. Eine konkrete Anwendungsstruktur liegt dann in Form eines XML-Dokumentes vor, das für die entsprechende DTD *gültig* ist. Die DTD entspricht dann einem *Architekturstil* im Sinne von

[Klein & Kazman, 1999] als Beschreibung von Komponententypen und ihrer Topologie.

Durch die Verwendung von erweiterten Schnittstellenbeschreibungen in Form des vorgestellten Typmodells, das notwendige Details zur Interaktion und Verknüpfung bereitstellt, kann die Anwendungsarchitektur selber sehr abstrakt und einfach ausfallen. Dies entspricht der Grundidee der vorliegenden Arbeit: Eine fachnahe Anwendungsmodellierung soll mittels eines direkten Übergangs durch Automatismen zu einer konkret implementierten Applikation führen, wobei die »fehlende« (den Übergang untersützende) Information direkt den einzusetzenden Komponenten (bzw. der zu ihnen gehörigen Meta-Informationen) entnommen werden soll.

Die verwendeten Komponenten tragen eine erweiterte Schnittstellenbeschreibung in XML,[44] die auf den Komponententechnologie-unabhängigen Schnittstellenbeschreibungsstandards des CORBA Component Models [OMG, 1999] aufbaut. Damit werden für die Beschreibung der Komponenten und der Anwendungsstruktur verwandte Techniken eingestzt, was die Realisierung der Generatormechanismen erleichtert und insbesondere ein *konstruktives Parsing* der zu verarbeitenden Informationen erlaubt.

Der Generator generiert eine Rahmenanwendung, die die benötigten Komponenten miteinander verknüpft, ansonsten jedoch keinen funktionalen Code umfasst. Es wird der Quellcode für das Anwendungsgerüst erzeugt, der vom Generator in der letzten Phase kompiliert und zusammen mit allen benötigten Komponenten in ein Produkt-Archiv verpackt wird, das einem *Deployment*, d.h. der späteren Festlegung der Einsatzkonfiguration, direkt zugänglich ist.

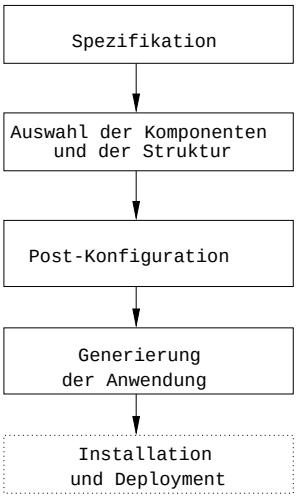
5.8.4 Phasen der Generierung

Die Generierung einer Anwendung erfolgt in mehreren Phasen, die dabei zwar konzeptionell einzeln benannt werden können, sich jedoch im konkreten Generator – bzw. der eigentlichen Anwendungskonstruktion – durchaus überschneiden können.

In der ersten Phase werden die Merkmale der gewünschten Anwendung ausgewählt. Auf Grund der ausgewählten Merkma-

[44] Siehe hierzu die konkrete Implementation des Typmodells in Abschnitt 6.2.

Abbildung 5.7
Phasen der Generierung



le werden dann die benötigten Komponenten[45]) ausgewählt und mit Hilfe der abstrakten Strukturbeschreibung (Architekturstil), in diesem Fall in Form einer DTD, in eine konkrete Anwendungsstruktur (Systemarchitektur) gebracht. Ein gegen die DTD validiertes XML-Dokument beschreibt diese Struktur mit den konkreten Komponenten der Anwendung und dient im Sinne eines abstrakten Syntaxbaums (vgl. a. Abschnitt 5.6.1) intern als Grundlage für die nächste Phase, die Post-Konfiguration. Diese XML-Repräsentation kann dabei zugleich wieder als Beschreibung der Gesamtanwendung im Sinne des gewählten Komponentenmodells interpretiert werden, sodass auch die entstehende Anwendung später als »korrekte« Komponente vorliegt und entsprechend weiterverwendet (in einer anderen Anwendungskonstruktion eingesetzt) werden kann. Aus allen Informationen wird in der letzten Phase die spezifisch konfigurierte Anwendung generiert. In einer weiteren Phase, dem Deployment, kann der Generator die Anwendung gleich in die Laufzeitumgebung installieren, da er auch die dazu notwendigen Informationen den eingelesenen Deskriptoren entnehmen kann.

[45] Die entweder direkt diesen Merkmalen entsprechen oder durch sie gekennzeichnet sind, die entsprechenden Werkzeuge des Kapitels 6 unterstützen beide Ansätze.

5.8.5 Verteilung der Informationen

Die Informationen, die zur Anwendungserstellung benötigt werden, entstammen verschiedenen Quellen. Bei der traditionellen Programmierung ist der Anwendungsentwickler oft alleine für die Anwendungsstruktur, sowie alle anderen Programmierdetails zuständig. Wenn ein Framework eingesetzt wird, muss der Entwickler den Code an seine Bedürfnisse anpassen. Bei der generativen Konstruktion sollten möglichst viele Operationen automatisch ablaufen und der Entwickler sollte sich nicht um Implementierungsdetails kümmern müssen, sondern um die Spezifikation der gewünschten Anwendung. Implementierungsdetails werden durch den Generator und die vorhandenen Komponenten bestimmt.

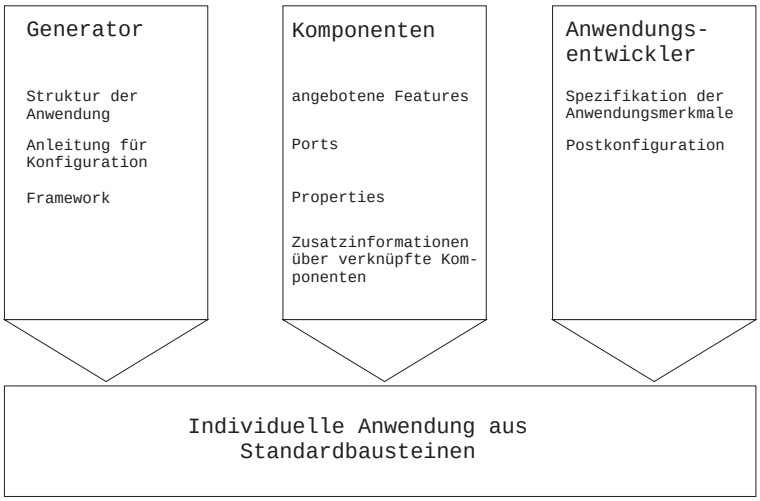


Abbildung 5.8
Quellen des Wissens zur Generierung

Entspechende Informationsquellen sind hier:

- *Generator* – Der Generator ist für die Strukturierung der Anwendung zuständig. Es ist ausserdem notwendig, den Anwender durch die Konfiguration der Anwendung zu führen und dabei wichtig, ihn durch eine zum Teil subjektiv, zum Teil problemspezifisch vorgegebenene Abfolge der vorzunehmenden Auswahlentscheidungen zu leiten. Diese Abfolge ist ebenso wie die Anwendungsstruktur domänenspezifisch und muss dementsprechend für einen Meta-Generator austauschbar sein.
- *Komponenten* – Es werden die Typbeschreibungen der Komponenten eingelesen und ausgewertet. Sie bestimmen, wel-

che Merkmale für die einzelnen Komponenten zur Auswahl stehen. Ausserdem werden zusätzliche Informationen ausgewertet, die beim Generieren des eigentlichen Anwendungsgerüstes benötigt werden, so zum Beispiel Angaben über Ein- / Ausgabe-Ports u.a. (vgl. a. Kapitel 2), die als Verknüpfungspunkte dienen. Eine Komponente kann gewisse Eigenschaften von mit ihr zu koppelnden Bausteinen beinflussen bzw. festlegen (vgl. a. Abschnitt 3.10). Die Typbeschreibungen enthalten gemäß des Modells aus Kapitel 4 auch Informationen für die Kompositionsvalidierung (vgl. a. Abschnitt 5.7.1.2).

- Anwendungsentwickler – Der Softwarekonstrukteur bestimmt, welche Art von Anwendung generiert wird. Er bestimmt die Merkmale der Anwendung und kann das Ergebnis zusätzlich durch eine Post-Konfiguration anpassen. Dies umfasst jedoch keine technischen Implementierungsdetails, für die ausschließlich der Generator und die verwendeten Komponenten zuständig sind.

Die entscheidenen Eingaben an Domänenwissen in den Generator sind die Beschreibung der Anwendungsstruktur, aus der die einzusetzenden Komponenten abgeleitet werden und deren Deskriptoren, aus deren Information zu Verknüpfungspunkten und zugehörigen Randbedingungen das Anwendungsgerüst abgeleitet wird. Die Deskriptoren stellen auch die Basis zur Kompatibilitätsprüfung einzusetzender Komponenten dar und lassen eine Assoziation mit der gewünschten Anwendungsstruktur zu. Mit Hilfe der Informationen aus den Deskriptoren ist es auch während der Konstruktion auch möglich gemäß des Konzeptes aus Kapitel 3 Adapter zu generieren, die der Generator im Rahmen des Anwendungsgerüsts vorsehen kann.

5.9 Komponentendeskriptoren: XML als Ausdrucksmittel

XML ist eine sehr vielseitige Beschreibungssprache für Komponentenspezifikationen, da es sehr einfach möglich ist, eigene Beschreibungselemente hinzuzufügen, sodass beispielsweise neben den üblichen Operationssignaturen auch andere der bereits geforderten Informationen wie etwa Randbedingungen und Einsatzregeln mit einem durchgängigen Ausdrucksmittel dargestellt werden können. Schnittstellenbeschreibungen von Komponenten in XML haben auch den Vorteil plattformunabhängig ausge-

wertet werden zu können. Es haben hier erste Standardisierungen stattgefunden (vgl. etwa [van Hoff et al., 1997], [OMG, 1999] und [Sun, 2000]), sodass ein pragmatischer Ansatz diese berücksichtigen sollte, damit entsprechende Bescheibungen auch in üblichen (kommerziellen) Entwicklungsumgebungen (weiter-) verwendet werden können, um nicht eine weitere dann möglicherweise ungenutzte akademische Sprachsyntax einzuführen. Gegenüber von sprachgebundenen Meta-Informationskonzepten wie etwa der Introspektion bzw. Reflection in Java [Griffel, 1998], hat die Verwendung Code-externer *Deskriptoren* den Vorteil einerseits technologieneutral zu sein und andererseits auch Konzepte wie etwa Vor- / Nachbedingungen u.ä. beschreiben zu können, die sich in konkreten Implementationssprachen unter Umständen nicht finden.

Deskriptoren tragen in modernen Systeminfrastrukturen für Komponenten zur Integritätskontrolle und zum Konfigurationsmangement einer Software erheblich bei [Broschei, 2000]. Ebenso wird die oftmals notwendige Vorwegnahme einer angenommenen späteren Einsatzumgebung beim Kunden vermieden, da die Deskriptoren eine deklarative Anpassung an die spezifischen Gegenbenheiten der konkreten Zielumgebung erlauben (sogenanntes Deployment) [Sun, 2000]. Daher sollen sowohl die konkreten Realisierungen des Typmodells aus Kapitel 4 als auch die weiteren im Rahmen einer Generator-gestützen Verarbeitung notwendigen Informationen in Form solcher Deskriptoren umgesetzt werden. Im folgenden sollen nur die grundlegenden Konzeption betrachtet werden, die Details entsprechender Deskriptoren und ihre konkrete Syntax werden im nachfolgenden Kapitel 6 erörtert.

5.9.1 Deskriptoren zur Schnittstellenbeschreibung

Während die Ausdrucksmöglichkeiten ursprünglicher Deskriptorenansätze wir etwa das Open Software Description (OSD) Format [van Hoff et al., 1997] sehr beschränkt waren, bieten die aktuellen Beschreibungen etwa des CORBA Component Modells (die ihrerseits auf OSD beruhen) weit mehr Möglichkeiten. Diese Beschreibungen wurde als Grundlage der Gestaltung der Deskriptoren in dieser Arbeit gewählt und entsprechend der Anforderungen des Klassifikations- und Generatorkonzepts erweitert, da sie für sich schon sehr flexibel und technologieneutral gehalten sind. Auf diese Weise bleiben gemäß den Ansätzen der vorliegenden Arbeit konkret implementierte Komponenten auch kom-

patibel zu gängigen industriellen Software- und Entwicklungsumgebungen.

Eines der Ziele der Object Management Group (OMG) mit dem CORBA Component Model (CCM) ist ohnehin die vereinfachte Verwendung von Komponenten in integrierten Entwicklungsumgebungen. Das Entwicklungswerkzeug kann auf diese Weise viele Informationen über Komponenten erhalten und die Arbeit für den Entwickler vereinfachen. Für die automatische Verwendung von Komponenten durch einen Generator sind allerdings weitere Informationen nötig. Der Generator benötigt folgende zusätzliche Informationen:

- Komponententyp (zur Auswahl, Suche von konkreten Bausteinen und der Einordnung in die Architektur)
- Merkmale (Features) der Anwendung für die Spezifikation und den Bezug zu Komponentenmerkmalen
- Properties (Eigenschaften) für ein zusätzliche Konfiguration bzw. verfeinerte Parametrisierung der Einzelkomponenten
- Properties, die Komponenten-übergreifend wirksam sind
- Mechanismus zur Verarbeitung visueller Komponenten[46]

Die CCM-Schnittstellen berücksichtigen keine explizite Klassifizierung von Komponenten. Eine solche erfolgt nur implizit über die angebotenen Schnittstellen. Dies ist für eine Verwendung in Entwicklungsumgebungen, wo die Schnittstellen beim richtigen Einsatz einer Komponente helfen sollen, oft ausreichend. Ein Generator benötigt jedoch zusätzliche semantische Informationen, aus denen etwa deutlich wird, welche Aufgabe diese Komponente in einer Anwendung übernehmen kann. Daher war die Konzeption eines entsprechenden Typkonzepts unabdingbar.

Ebenso werden im CCM-Modell keine Merkmale (im Sinne dieser Arbeit) der Komponenten berücksichtigt. Die Merkmale dienen beim Generator aber der Auswahl zwischen kompatiblen

[46] Dahinter steht die Problematik der automatischen bzw. wenigstens Schablonen-basierten Generierung von Benutzeroberflächen. Ein Ansatz, der eine *vollständig* automatisierte Erstellung einer Applikation fordert, müsste dies für entsprechend interaktive Anwendungen leisten, allerdings ist dies nahezu ausgeschlossen, da die kreativen Wünschen und Layout-Fähigkeiten eines menschlichen Entwickler fehlen. In wie weit etwa Ansätze in Form von »Layout-Managern« (etwa in Java [Spell, 2000]) entsprechend weiterentwickelt werden könnten, wurde im Rahmen dieser Arbeit nicht untersucht.

Komponenten und der Zuordnung zu gewünschten Anwendungseigenschaften. Auch für das Vorschlagen von Properties zu verwendender Komponenten (s. Abschnitt 6.3.6.4) müssen zusätzliche Beschreibungselemente eingeführt werden. Es ist ferner notwendig, Informationen einzuführen, die visuelle Komponenten unterstützen. Eine Beschreibung dieser hier als `GUISlots` bezeichneten Elemente muss ebenfalls in der Komponentenbeschreibung erfolgen, wenn die zur Anwendungskonstruktion notwendigen Informationen im wesentlichen auf den eingesetzen Komponenten selbst basieren sollen. In Abschnitt 6.3.7 wird an Hand einer Beipsielsspezifikation detailliert darauf eingegangen, wie die Deskriptoren gegenüber den Schnittstellenbeschreibungen des CCM-Modells erweitert wurden.

5.9.2 DTD zur Architekturrepräsentation

In [Neubert, 1998] erfolgte die Spezifikation einer Matrix in Form von Parameterwerten eines C++-Templates (vgl. Abschnitt 5.7.3.3). Dessen Auswertung wurde mit Hilfe weiterer C++-Templates in ein internes Format umgewandelt, das wiederum ein C++-Template mit Parameterwerten darstellt. Der eigentliche Aufbau der Matrix liegt damit in den C++-Templates versteckt. Bei GenVoca wurde eine klarere Schichtenarchitektur propagiert und der Aufbau einer Anwendung durch exakte Gleichungen bestimmt. Auch der hier zu implementierende Generator benötigt als Eingabe eine Architektur der zu generierenden Anwendung. Diese soll allerdings möglichst explizit und implementationsnah vorliegen, um einen entsprechend leichten und verlustfreien Übergang zwischen Modell und konkreter Applikation zu ermöglichen.

Es sind zwei Arten von Beschreibungen notwendig. Zum einen soll die Architektur einer Anwendungsdomäne ausgedrückt werden können. Dafür müssen die notwendigen Komponenten beschrieben werden können, sowie eine Struktur, die diese miteinander verknüpft. Zum anderen sollte es auch möglich sein, die *konkrete Struktur* einer *einzigen* Anwendung dieser Domäne zu beschreiben. Der Aufbau einer konkreten Struktur sollte gegen die Beschreibung der möglichen Strukturen getestet werden können.[47]

Die Struktur einer konkreten Anwendung sollte sich am hierarchisch kompositorischen Charakter des in Kapitel 2 zu Grunde

[47] Die *Menge möglicher* Anwendungsstrukturen definiert in dieser Arbeit den Begriff des *Architekturstils*. Eine konkrete Ausprägung eines solchen wird dann als Architektur bezeichnet.

gelegten Komponentenmodells orientieren. Eine Beschreibungssprache für strenge Hierachien würde Zyklen und damit verbundene Probleme vermeiden, schränkt allerdings die mögliche »Eleganz« der entstehen Modelle ein, was hier allerdings in Kauf genommen wird. Die Architekturbeschreibung sollte nicht in Form gängiger Architekturbeschreibungssprachen erfolgen. Die ADLs (s. Abschnitt 5.2.3) beschreiben neben der Struktur aus Komponenten und ihrer Interaktion untereinander auch semantische Eigenschaften der Anwendung. Durch den Einsatz entsprechend gemäß des vorgestellten Klassifikationskonzepts typisierter Komponenten stehen viele der benötigten Informationen schon (besser getrennt) zur Verfügung und müssen nicht in der Architekturbeschreibung wiederholt werden.[48] Der Generator kann die Typinformationen mit Hilfe eines Typmanagers auswerten, um einerseits die Konformität einer Komponente zur vorgesehenen Architektur – die letztlich eine strukturierte Sammlung von Komponententypen darstellt – zu überprüfen. Andererseits wird so die Kompatibilität mehrerer auf diese Weise ausgewählter Komponenten überprüft. Schließlich dienen die in den Typbeschreibungen enthaltenen Schnittstellenspezifikationen der Verknüpfung der ausgewählten und geprüften Komponenten. Die Architekturbeschreibung selbst kann daher relativ einfach und mit entsprechend fachnahem Vokabular gestaltet werden.

Mehreren Anforderungen sollte die benötigte Beschreibungssprache genügen. Sie sollte

- mögliche Strukturen einer Domäne beschreiben,
- eine einfache Übertragung auf die Beschreibung einer einzelnen Anwendung erlauben,
- ausdrucksstark und flexibel sein,
- streng hierachische Strukturen beschreiben,
- auf Standards zurückgreifen, sodass Werkzeuge zur Verfügung stehen,
- zur Repräsentation des Komponenten- bzw. Typmodells passen,
- Plattform- bzw. Programmiersprachen-unabhängig sein und
- einfach und übersichtlich in ihrer Syntax sein.

[48] Genau dies war ja das Ziel der Externalisierung des Typkonzepts.

Die Wahl fiel auf das Konzept der Document Type Definition (DTD) zur Beschreibung aller möglichen Anwendungsstrukturen einer Domäne. Eine konkrete Anwendungsstruktur liegt dann in Form eines einzelnen XML-Dokumentes vor, das die DTD erfüllt. Somit kann getestet werden, ob eine Struktur einer einzigen Anwendung für eine Domäne Gültigkeit besitzt. Die Elemente des XML-Dokumentes repräsentieren dabei die Namen von Komponententypen, d.h. jeweils einer Klasse von Komponenten, die geeignet wären eine entsprechende Position im Rahmen der Anwendungsstruktur einzunehmen.

Eine DTD bietet eine gute Grundlage, den Aufbau von Anwendungsstrukturen zu beschreiben. Generell werden Anwendungen einer Domäne im Aufbau ähnlich sein. Die DTD bietet Möglichkeiten wie die Definition von optionalen Elementen oder Entscheidungen zwischen 2 Arten von Elementen (Oder-Verknüpfung). Auch das Konzept von symmetrischen Komponenten (vgl. a. Kapitel 2 und Abschnitt 5.7.1) kann durch Deklarationen abgebildet werden. Eine Deklaration `A*` würde zum Beispiel bedeuten, das keine oder beliebig viele Komponenten vom Typ A an einer Stelle vorkommen können.

5.9.3 Klassifikation der Komponenten

In Analogie zu GenVocas Basiskategorien (Realms) werden Komponenten auch für den zu implementierenden Generator *klassifiziert*. Komponenten, die eine ähnliche[49] Funktionalität und Funktionsweise [Griffel, 1998] besitzen, werden unter einem Komponenten*typ* gemäß des Konzepts aus Kapitel 4 zusammengefasst. Die Typisierung kann dann zur Kennzeichnung der funktionalen Bereiche innerhalb einer Anwendungsarchitektur und so der späteren Zuordnung zu konkreten Komponenten dienen, wobei die Konformität zur ursprünglichen Spezifikation gewährleistet bleibt. Der Name der als domänenspezifische Sprache verwendeten XML-Elemente entspricht dabei später der jeweiligen Bezeichnung des Komponententyps. Da diese Typinformationen durch ein Typmanagement verwaltet und dem Generator übermittelt werden, ist es so insbesondere auch möglich, geeignete Komponenten auf Grund ihres konformen Verhaltens auszuwählen und nicht nur ihrer blossen (mehr oder weniger aussagekräftigen) Namen.

Der Komponententyp sagt aus, dass gewisse Schnittstellen von einer Komponente eingehalten werden und auch das semantische Verhalten ähnlich ist. Bei der Domäne »Taschenrechner«

[49] vgl. a. den Affinitätsbegriff in Abschnitt 4.4.2.1

gibt es z.B. den Komponententyp *Funktionseinheit*, der eine oder mehrere Berechnungen durchführen kann. Dabei kann es sich z.B. um eine Komponente für die Berechnung des Sinus oder um eine Komponente, die alle Grundrechenarten anbietet, handeln. Die Funktionweise dieser Komponenten weist Ähnlichkeiten auf, nämlich die Berechnung mathematischer Funktionen über Zahlen, sodass sie für einen Rechner sinnvoll einsetzbar sind.

5.9.4 Zusammenfassung

Zusammenfassend lässt sich die Grundkonzeption des Generators nochmals folgendermassen umreißen: Seine Aufgabe besteht darin, eine abstrakte Auswahl von Merkmalen einer Anwendung zu erlauben und für diese Auswahl die benötigten Komponenten für eine Anwendungsarchitektur zu suchen. In der Architekturbeschreibung stehen die Komponententypen, die notwendig sind, um die für die Komponenten vorgesehenen Plätze (slots)[50] der Anwendungsstruktur zu besetzen. Aus der Liste der dann gewählten Komponenten wird ein XML-Dokument konstruiert, das für die dazugehörige DTD, die die Architektur beschreibt, gültig (validiert) ist. Aus diesem XML-Dokument und den Komponentenschnittstellen, die im Rahmen der eingesetzten Deskriptoren ebenfalls in XML vorliegen, kann der Generator eine korrekte Anwendung generieren. Die Beschreibung von allgemeinener und konkreter Anwendungsstruktur, sowie die Beschreibung der Funktionalität liegen also als XML-Repräsentationen vor. Der Generator muss auf diese Weise nur eine einzige Syntax in Form eines konstruktiven Parsing verarbeiten (s.a. Abschnitt 6.3.5).

5.10 Auswahl der Komponenten

5.10.1 Auswahl der Merkmale einer Anwendung

Die Auswahl der Komponenten soll durch die Zuordnung der die Semantik ihres Interaktionspotenzials definierenden Merkmale (vgl. Kapitel 4) zu den gewünschten Anwendungsmerkmalen erfolgen. Diese geforderten Merkmale – also letztlich die Anwendungsspezifikation – soll hierbei *angeleitet*, beispielsweise durch einen graphischen »Wizard«, erfolgen, um etwa unmögliche Konfigurationen von vornherein auszuschließen. Die Merkmale, die dabei dem Konstrukteur zur Auswahl angeboten werden, entstammen aber nur indirekt aus der ursprünglichen Domänen-

[50] vgl. a. das *hot-spot*-Konzept für Rahmenwerke in [Pree, 1997]

analyse. Sie sollen vielmehr direkt von den *wirklich zur Verfügung stehenden* Komponenten abgeleitet werden.[51] Daher wertet der Generator die Typen der vorhandenen Komponenten aus, in denen ihre Merkmale definiert sind. Mit Hilfe dieser Merkmale kann ein Komponentenhersteller etwa betonen, wodurch sich gerade seine Komponente besonders auszeichnet und weshalb jemand gerade sie auswählen sollte. Es ist auch möglich nichtfunktionale Zusatzinformationen zu berücksichtigen. So könnte letztendlich der Preis einer Komponente oder ihr Ressourcenbedarf bei der Wahl zwischen ähnlichen Komponenten eines Types zur Entscheidungsfindung beitragen. Zusätzlich besteht die Möglichkeit zu jedem Zeitpunkt weitere Ausprägungen eines Komponententypen hinzuzufügen, da die möglichen Merkmale einer *individuellen* Anwendung nicht durch die entsprechende Aufbereitung der Domäne schon fest vorgeschrieben werden. Merkmale, die die Auswahl mehrerer Komponenten gleichzeitig betreffen, sind möglich, da sie entsprechend in der Domänenanalyse standarisiert wurden (vgl. a. Seite 290).

Neben der Abfrage der gewünschten Merkmale muss der Softwarekonstruteur während des Generatoreinsatzes zusätzlich Entweder-Oder-Entscheidungen treffen, die die konkrete Struktur der Anwendung beeinflussen. Diese werden auf Basis der Priorisierung der Auswahlentscheidungen abgefragt (s. Abschnitt 5.10.4).

5.10.2 Kompatibilität von Komponenten

Um eine bessere Abgrenzung zu den Eigenschaften (»Properties«, »Attributes«) eines konkreten Komponentenmodells (wie etwa den im Prototypen eingesetzen JavaBeans) zu gewährleisten, wird von den Auswahlmerkmalen im weiteren als *Features* einer Komponente gesprochen. Die Repräsentation eines Features erfolgt über entsprechende Merkmalsstrukturen[52] und erhält zudem einen Featurenamen und einen Featurewert zugeordnet. Das Feature einer trigonometrischen Funktion könnte zum Beispiel mit `Funktionsart=sinus` gekennzeichnet werden. Die gewählten Bezeichnungen sollten – zumindest für Fachleute eines Anwendungsbereichs – selbsterklärend sein und dem Vo-

[51] In diese ist natürlich das Domänenwissen eingeflossen. Eine entsprechend direkte »Abfrage« erlaubt jedoch den Verzicht auf zusätzliche »Wissensbasen« o.ä. und entsprechend dem Komponentengedanken im Sinne des Tragens der eigenen Meta-Information.

[52] Die wiederum als Terme der in Kapitel 2 dargestellten Logik auftreten können.

kabular der jeweils betrachteten Domäne entstammen (vgl. Abschnitt 5.3).

Die Featurenamen müssen speziell genug aus dem Domänenvokabular gewählt werden, um ungewollte Überschneidungen zu vermeiden, aber allgemein genug um Abhängigkeiten und Gegebenheiten des jeweiligen Anwendungsbereichs auszudrücken. Während zum Beispiel das Feature `Externes_Zahlensystem=dezimal` sicherlich Sinn macht, damit einzelne Komponenten von der gleichen Zahlenrepräsentation ausgehen, wäre das Feature `Farbe=blau` zu allgemein. Sinnvoller wäre es, letzteres Feature zum Beispiel situationsbedingt `Hintergrundfarbe=blau` zu nennen.

Die Zuordnung entsprechender Namensbezeichner ist unabdingbar, da der menschliche Konstrukteur später fachspezifische Konzepte nicht an Hand unter Umständen komplexer Bäume von Merkmalsstrukturen identifizieren kann.[53] Die unmissverständliche *Bedeutung* ist jedoch über die entsprechenden Strukturen festgelegt und kann bei Bedarf überprüft werden (vgl. a. Kapitel 4).

Es ist ein Mechanismus notwendig, der sicherstellt, dass Features, die die Auswahl mehrerer Komponententypen betreffen, auch von *allen* diesen Komponenten erfüllt bzw. geboten werden. Ein bei einem vorher zu spezifizierenden Komponententyp ausgewähltes Merkmal, kann damit die Auswahlmöglichkeiten bei einem anderen Komponententyp einschränken. Es wird daher die Bezeichnung *signifikant* für Features eingeführt, die durch die Spezifikation vorheriger Komponententypen automatisch ausgewählt werden und die bei der Auswahl einer weiteren Komponente eines Types zu berücksichtigen sind. Diese Features können dabei direkt vom Anwender oder indirekt durch die Komponentenspezifikation bestimmt worden sein. Indirekt ausgewählte Features sind Merkmale, die eine Komponente anbietet, die aber nicht in der Spezifikation durch den Anwender ausgewählt worden sind. Da diese Features aber fester Bestandteil der ausgewählten Komponente sind, müssen sie auch im weiteren Ablauf der Konfiguration berücksichtigt werden. Die Merkmale erfüllen damit neben den Policies (vgl. Abschnitt 3.10) eine ähnliche Aufgabe wie die Conditions und Restrictions im GenVoca-Modell (vgl. Abschnitt 5.7.1.2). Sie haben dabei jedoch einen globalen Charakter im Sinne eines »cross-cut« durch die gesamte Applikation (vgl. Abschnitt 5.5.1.1).

[53] vgl. etwa a. das Konzept des »Display-Name»in [Sun, 2000]

Es sind dabei aber nicht die Merkmale aller vorher ausgewählten Komponenten zu berücksichtigen. Es werden nur die Merkmale berücksichtigt, mit denen höchstens eine Komponente des jeweiligen Types für die Anwendung ausgewählt werden darf. Wenn ein Komponententyp mehrfach innerhalb der Anwendungsarchitektur auftritt, widersprächen sich die Featurewerte entsprechender konkreter Komponenten unter Umständen. Zwei Komponenten des gleichen Typs unterscheiden sich in der Regel, zum Beispiel in mindestens einem Featurewert. Wenn zwei Komponenten die exakt gleichen Featureswerte anbieten, dann haben sie auch die exakt gleiche Funktionalität und unterscheiden sich höchstens in der technischen Implementierung. Funktionseinheiten (vgl. Abschnitt 5.12) können sich zum Beispiel in dem Feature `Methodenname` unterscheiden. Die eine Funktionseinheit bietet das Feature `Methodenname=sinus`, während eine andere für den gleichen Featurenamen `Methodenname` den Wert `cosinus` und damit eine unterschiedliche Funktionalität anbietet. Wäre es möglich, sowohl eine Komponente auszuwählen, die das Feature `Methodenname=cosinus` anbietet, als auch eine, die das Feature `Methodenname=sinus` spezifiziert hat, müssten andere Komponenten bei einer darauf folgenden Auswahl einer Komponente eines anderen Types beide Features implementieren. Dies ist aber nicht möglich, da eine Komponente immer nur *einen* Wert für einen Featurenamen definieren kann. Es kann auch keiner der beiden Werte priorisiert werden.[54]

Es werden bei der Auswahl eines Komponententyps allerdings nicht alle Features, die in vorangegangenen Spezifikationen, direkt oder indirekt, ausgewählt wurden, berücksichtigt. Features sind nur dann signifikant für einen Komponententyp, wenn mindestens eine Komponente dieses Types ein Feature mit diesem Featurenamen in ihrer Spezifikation definiert hat. Der Generator geht dann davon aus, dass dieses Feature eine Bedeutung für diesen Komponententyp hat. Die signifikanten Features nehmen Einfluss auf die Kopplungsmöglichkeit der Komponenten untereinander.

[54] Dieses Beispiel aus der später gezeigten Umsetzung eines Taschenrechners ist bewusst künstlich und minimal, um verschiedene Aspekte und Probleme diskutieren zu können. Dies impliziert natürlich nicht, dass es nicht denkbar wäre, Komponenten als Komposita mehrerer Funktionen und entsprechend Methoden vorzusehen.

5.10.3 Verknüpfung von Komponenten

Im zu Grunde gelegenten Komponentemodell des Kapitels 2 wurde eine streng hierarchische Anwendungsstruktur gefordert. Die Anwendung kann somit als Baum mit den Komponenten als Blättern dargestellt werden. Nur Komponenten die innerhalb dieses Baums in einer Vater-Sohn-Beziehung stehen, sollen miteinander verknüpft werden können. Die Verknüpfung erfolgt konzeptionell über das dem adaptiven Typmodell zu Grunde gelegten Nachrichtenmodell.

Eine Komponente gibt das ihr eigene Interaktionspotenzial über ihrem Typ an. Wenn der Generator für zwei während der Konstruktion zu koppelnde Komponenten mittels des Typmanagements kompatible Komponenten vorfindet, wertet er die syntaktische, technisch orientierte Ebene der zugehörigen Typbeschreibungen aus, um eine entsprechende Verbindung durch das erstellte Anwendungsgerüst herzustellen. Da hierbei technisch wiederum unterschiedlichste Ansätze zum Einsatz kommen können (etwa die Kopplung über Eignisquellen / -senken, Publish / Subscribe-Verfahren nachrichtenorientierter Middleware, entfernte Prozeduraufrufe etc., vgl. a. [Griffel, 1998]), muss auch dieser Teil des Generators rekonfigurierbar sein (d.h., selbst als austauschbare Komponente vorliegen). Zur Verdeutlichung dieses Vorgehens und des Nachweises seiner Praktikabilität wurden im Rahmen der vorliegen Arbeit exemplarisch drei solche Verknüpfungsmodule erstellt, je eines zur Kopplung mittels lokaler Ereignisse von JavaBean- und COM-Komponenten und eines zur verteilten Kopplung mittels des Java Messaging Services (JMS) (s.a. Abschnitt 6.3).

Motiviert ist die entsprechende Vorgehensweise durch das (derzeit insbesondere von Sun im Java Umfeld) sehr erfolgreich eingesetzte Konzept der *insulation*, also der Kapselung durch generische Zugriffsschnittstellen.[55] Die Art der Kopplung sollte dabei möglichst technologieneutral sein, d.h. Adapter bzw. Interzeptionskonzepte in Form von Konnektoren sollten generisch betrachtet werden (also nicht etwa ein spezifischer CORBA-Interceptor im ORB [Griffel, 1998] etc. eingesetzt werden). Die technische Abbildung der Konzepte ist dann natürlich technologieabhängig.

[55] Sehr erfolgreiche Praxisbeispiele sind für die Bereiche Datenbanken: ODBC, Namens- und Verzeichnisdienste: JNDI, Transaktionsmonitore JTA etc.

5.10.4 Priorisierung der Auswahlentscheidungen für Merkmale

GenVoca (s. Abschnitt 5.7.1) zeigt eine Schichtenarchitektur, bei der die obere Schicht von der jeweils darunter liegenden parametrisiert wird. Durch die Festlegung der Schichten wird dort die Reihenfolge der Entscheidungsfindung bestimmt. Eine solche Reihenfolge bei der Auswahl von Komponenten ist notwendig, da Abhängigkeiten zwischen einzelnen Komponenten bestehen können, sobald diese gemeinsam eingesetzt werden. Die Auswahl einer Komponente kann die spätere Auswahlmöglichkeit weiterer Komponenten wie schon gesehen einschränken. Hierbei müssen die in der Domänenanalyse identifizierten Wechselwirkungen berücksichtigt werden. Es muss dabei der Grad der *Bedeutsamkeit* einer Komponente für eine bestimmte Anwendungsart berücksichtigt werden. Analog verfeinert ein Parameter, der in einer GenVoca-Gleichung weiter innen spezifiziert wird, dabei immer die bisher getroffene Auswahl der äusseren Parameter, kann diese aber nicht mehr aufheben.

Das Beispiel des Autos verdeutlicht dies nochmals: Beim Kauf eines Autos bietet ein Hersteller verschiedene Modelle an, für die unterschiedliche Motoren zur Auswahl stehen. Es paßt dabei aber nicht jeder Motor zu jedem Modell. Ein Kunde wird normalerweise erst das Modell aussuchen und davon abhängig einen der möglichen Motoren. Der Motor spielt eine sekundäre Rolle. Unter eine anderen Priorisierung, beispielsweise eines sportlichen Fahrens, könnte aber tatsächlich auch, zunächst ein leistungsstarker Motor gewählt werden.

Eine ähnliche Flexibilität der Reihenfolge der Entscheidungen für die Anwendungspezifikation im Rahmen einer angeleiten Konfiguration ist notwendig, sollte jedoch unabhängig von den identifizierten möglichen Architekturen sein. Daher wird eine zusätzliche Beschreibung zur Steuerung der angeleiteten Auswahl der Komponententypen in der ersten Generierungsphase eingeführt. Auch diese Beschreibung liegt in Form einer XML-Datei vor (s.a. Abschnitt 6.3.2.1). Die Reihenfolge der abgefragten Entscheidungen ist allerdings nicht starr, da der Anwender eventuell Entscheidungen zwischen mehreren Komponententypen treffen muss, die den weiteren Verlauf der Spezifikationsphase bestimmen. Auf diese Weise kann es auch in der Struktur der generierten Anwendungen zu Unterschieden kommen. Gleichzeitig bietet dieser strukturelle Freiheitsgrad Optimierungsmöglichkeiten für die Konstruktion (s.a. Abschnitt 7).

5.11 Leitbild für die Implementation

Im Folgenden wird ein Leitbild für eine die obigen Konzepte realisierende Implementation und deren Einsatz gegeben, deren technische Details dann in Kapitel 6 vorgestellt werden. Dabei soll der typische Ablauf bei der Konstruktion konzipiert und illustriert werden, da dies dem Entwicklungsprozess bzw. dem *Vorgehensmodell* des Ansatzes entspricht.

5.11.1 Der Generator aus Anwendersicht

Der Generator startet mit einem Fenster, das es zunächst erlaubt einzustellen, welche Art von Anwendung (Produktlinie) erstellt werden soll, welches Komponentrepository Bausteine und Schnittstellen bereitstellt, sowie den Namen für die zu generierende Anwendung. Wenn diese Einstellungen vorgenommen wurden oder entspechende Standardeinstellungen akzeptiert werden, kann der eigentliche Generierungsprozess gestartet werden.

Für den Aufbau der graphischen Benutzerführung zur Konfiguration werden die Deskriptoren der vorhandenen Komponenten eingelesen und ausgewertet.[56] Der Anwender wird nach einer während des Domain Engineering bestimmten Priorisierung (Reihenfolge) für jeden in der eingelesenen Architekturspezifikation auftretenden Komponententyp aufgefordert, gewünschte Merkmale auszuwählen. Die zur Auswahl stehenden Merkmale hängen somit von den vorhandenen Komponenten ab.[57] Bei dieser Auswahl wird eine Unterscheidung vorgenommen, wie viele Komponenten eines Typs in der zu erstellenden Anwendung vorkommen dürfen. Es sind die folgenden Fälle zu unterscheiden:

- Es *muss genau eine* Komponente des Komponententypes ausgewählt werden (DTD-Repräsentation: A).
- Es *kann genau eine oder aber keine* Komponente des Komponententypes ausgewählt werden. Der Komponententyp ist damit optional (DTD-Repräsentation: A?).

[56] Es werden dabei nur Komponenten aus jeweils genau einer Domäne berücksichtigt, da ein übergreifender Einsatz zu semantischen Inkonsistenzen führen könnte.

[57] Die Gefahr einer potenziellen Vermischung von Analyse- und Designphase wie sie in der bottom-up Entwicklungsweise in der allgemeinen Komponentesicht besteht (der Entwickler lässt sich von bestehenden Komponenten leiten, statt problemspezifisch zu Entwerfen und *dann* Implementationslösungen zu suchen) ist hierbei deutlich gemindert, da die vorgegebene Anwendungsarchitektur als Ergebnis einer Domänenanalyse eine sorgfältige Problemanalyse impliziert.

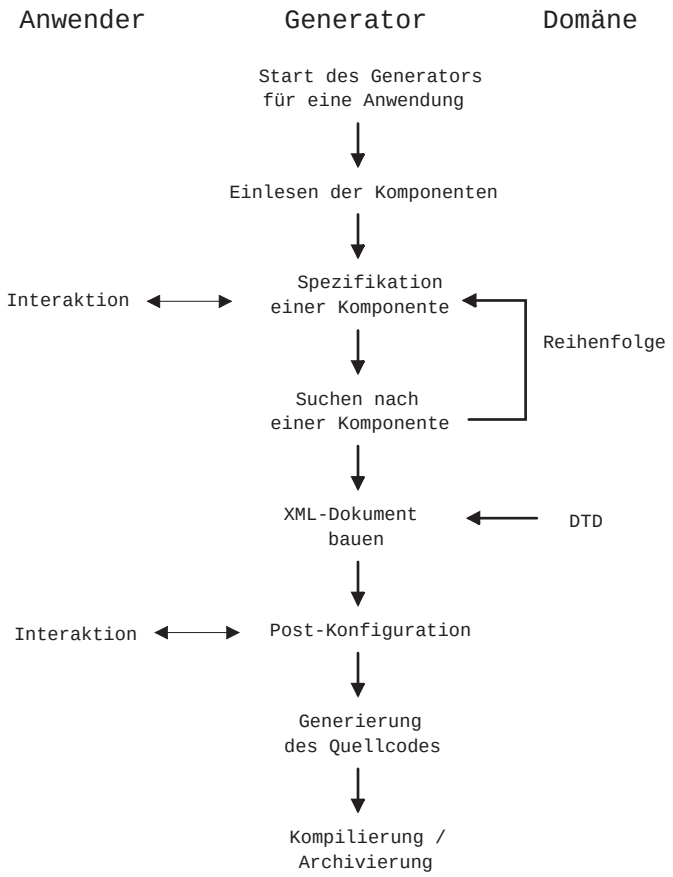


Abbildung 5.9
Der Ablauf eines Generierungsvorganges

- Es können *beliebig viele* Komponenten des Komponententypes ausgewählt werden (DTD-Repräsentation: A*).
- Es können *beliebig viele, aber mindestens eine* Komponente des Komponententypes ausgewählt werden (DTD-Repräsentation: A+).

Der Generator wiederholt die Auswahl von Merkmalen für einen Komponententyp, der mehrfach vorkommen kann, bis der Anwender diese Auswahl abbricht. Bei Typen, die vorkommen können, aber nicht müssen hat der Anwender vorher die Möglichkeit zur Auswahl des nächsten Typs zu springen. Es können auch Entweder-Oder-Entscheidungen zwischen verschiedenen Komponententypen getroffen werden, wobei der Generator (bzw. der zugehörige Konfigurationswizard) dem Anwender die Konsequenzen der Entscheidung erläutert.

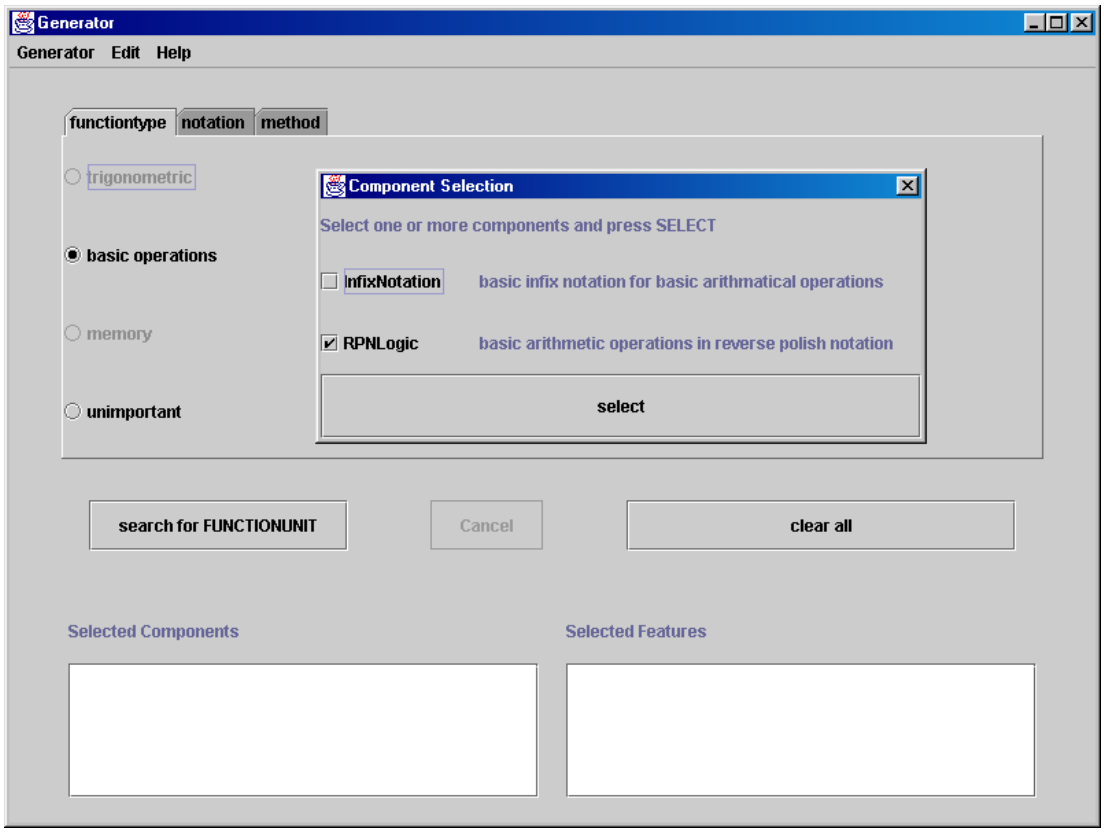


Abbildung 5.10
Der Aufbau des grafischen Wizards

Der Konstrukteur bekommt in einem Fenster mehrere Karteikarten zu sehen (s. Abbildung 5.10), wobei jedes Merkmal mit den möglichen Werten, die zur Auswahl stehen, eine eigene Karte darstellt. Standardmässig wird vor ausgewählt, dass ein Merkmal bei der Auswahl der Komponente *nicht* berücksichtigt werden soll.[58] Der Anwender kann beliebig zwischen den verschiedenen Karteikarten hin- und herspringen, also sich frei für Merkmalskombinationen entscheiden. Ändert der Anwender allerdings die Auswahl eines Merkmals, wird dafür gesorgt, dass nur noch jeweils die Werte von Merkmalen ausgewählt werden können, die eine gültige Kombination darstellen, d.h., dass zumindest eine Komponente im Komponentenpool (Re-

[58] Somit wird sichergestellt, dass auch ein unkundiger oder unter Zeitdruck entscheidender Konstrukteur konsistente Anwendungen erstellt, indem die vom Hersteller einer Komponente vorgesehenen Standardeinstellungen (defaults) genutzt werden, die im Idealfall für die »plug&play«-Fähigkeit einer Komponente sorgen.

pository) vorliegt, die alle ausgewählten Merkmale anbietet. Die Auswahl kann wieder rückgängig gemacht werden, indem der Standardwert ausgewählt wird. Ausserdem besteht die Möglichkeit, durch Betätigung einer Schaltfläche alle Merkmale wieder auf den Standardwert zurück zu setzen.

Die zur Auswahl stehenden Merkmale hängen wie erwähnt von den tatsächlich vorhandenen Komponenten ab, sodass bei einem entsprechend grossen Pool konformer Komponenten die Auswahlmöglichkeit automatisch grösser wird. Über eine Schaltfläche zur Suche startet der Anwender die Auswahl (bzw. Suche nach) einer konkreten Komponente. Bei Komponententypen, die mit mehrfachen Instanzen innerhalb einer Anwendungsstruktur auftreten, besteht die Möglichkeit, mehrere Komponenten gleichzeitig auszuwählen. Es erscheint ein Fenster, in dem alle Komponenten, die die ausgewählten Features berücksichtigen, aufgeführt sind. Der Anwender kann eine beliebige Anzahl direkt auswählen. Ansonsten wird er darüber informiert welche Komponente, die seinen Angaben entspricht, automatisch ausgewählt wurde (zum konkreten Verhalten vgl. a. Abschnitt 6.3).

Wenn die Anwendung mehrere Komponenten eines Komponententypes beinhalten kann, wird der Anwender nach jeder ausgewählten Komponente gefragt, ob er eine weitere Komponente dieses Typs bestimmen möchte. Bei optionalen Komponententypen wird er vor dem Aufbau der zugehörigen Karteikarten befragt, ob er eine solche Komponente überhaupt näher spezifizieren möchte. In beiden Fällen kann durch einen Schaltfläche zum Abbruch zur Bearbeitung zum nächsten Komponententyp gesprungen werden.

Auf diese Weise wird der Anwender durch die Spezifikation aller benötigten Komponenten der Anwendung geleitet.[59] Sind alle Komponenten gefunden, wird die eigentliche Anwendung konstruiert. Der Konstrukteur hat die Möglichkeit, eine weitere Anpassung (Individualisierung) der Applikation vorzunehmen, indem er in einer Post-Konfigurationsphase einzelne Eigenschaften der verwendeten Komponenten nachträglich setzt. Dies betrifft dann Eigenschaften einer Komponente, die keinen Einfluss auf die Auswahl der Komponente für die Anwendungsstruktur, d.h. keine strukturelle Wirkung hatten.[60] Der Generator bietet aber auf Basis der Deskriptor-Informationen sinnvolle Vorschläge für die Eigenschaften, die vom Anwender einfach übernom-

[59] Wird dabei generell die Standardeinstellung akzeptiert entspricht dies einer vollständig automatischen Auswahl.

[60] vgl. a. die Unterscheidung struktureller und funktionaler Systemeigenschaften in Abschnitt 5.5.2

men werden könnten. Danach stellt der Generator die eigentliche Anwendung aus den ausgesuchten Komponenten durch Generierung eines Anwendungsgerüstes zusammen.

5.12 Eine Beispieldomäne

Als exemplarischer Anwendungsbereich zur Demonstration des Generatoreinsatzes soll im weiteren die Anwendungsfamilie »Taschenrechner« dienen. Da Methoden und Verfahren der Domänenanalyse nicht Gegenstand dieser Arbeit sind, hat auch die entsprechende Domänenmodellierung nur illustrativen Charakter[61] und ist nicht als komplette Aufarbeitung des Anwendungsbereichs zu verstehen, soll aber typische Gedankengänge und Entscheidungen verdeutlichen. Es wird etwa nicht die Frage der Vollständigkeit eines Domänenmodells erörtert, da der vorgestellte Ansatz erst nach einer erfolgten Domänenmodellierung einsetzt und bereits vom Vorliegen einer Architekturbeschreibung ausgehen. Es müssen also solche möglichen Anwendungsstrukturen gefunden werden, die streng hierachisch aufgebaut und durch eine DTD ausgedrückt werden. In Abbildung 5.11 ist die Struktur eines Beispieltaschenrechners gezeigt.

Bei der Domänenanalyse wird hier nicht eine ganze Anwendungsfamilie (etwa mathematischer Anwendungen[62]), sondern hieraus die spezielle Produktlinie der Taschenrechner berücksichtigt. Es muss also untersucht werden, wie sich einzelne Taschenrechner unterscheiden können und welche Gemeinsamkeiten sie haben. Die einzelnen erkannten Funktionalitäten müssen aufgeteilt werden, sodass einzelne Komponenten identifiziert werden können. Zudem müssen die auftretenden Interaktionsmuster analysiert bzw. festgelegt werden, mit Hilfe derer die Komponenten miteinander kommunizieren. Die dabei auftretenden Nachrichten müssen ebenso wie die verwendeten Komponententypen beschrieben werden, um eine konsistente und kompatible Interaktion zu ermöglichen. Am Ende der Analyse steht eine Beschreibung der Anwendungsstruktur in Form einer DTD. Die Festlegung eventueller Prioritäten einzelner Komponenten

[61] Zum Teil sind getroffene Entscheidungen »künstlicher« Natur, in dem Sinne, dass sie nur gewählt wurden, um möglichst alle auftretenden Fragestellungen bei der Konzeption eines Generators auch konkret untersuchen bzw. veranschaulichen zu können.

[62] Dieses Konzept wurde im Rahmen der vorliegenden Arbeit durch die Betrachtung der weiteren Produktlinien »Formeleditoren« und »Graphenplotter« ebenfalls in der Praxis verdeutlicht

ermöglicht später eine gewichtete Auswahl von Merkmalen. Die Festlegung von Entweder-Oder-Entscheidungen oder Optionen ermöglichen eine Architektur mit entspechende hoher Variabilität, um später unterschiedliche Ausprägungen des Taschenrechners (eben eine Produkt*linie*) zu erhalten.

5.12.1 Unterscheidungsmerkmale von Taschenrechnern

Eine Applikation »Taschenrechner« kann sich in ihrem Aussehen, ihrer Bedienung und in ihrer Funktionalität unterscheiden. Es gibt unterschiedlich grosse Anzeigen und das Aussehen des Bedienfeldes (Tasten) kann unterschiedlich ausfallen. Auch das »Gehäuse« kann verschiedene Formen und Farben besitzen, wobei zum Beispiel die Farben Gegenstand einer Post-Konfiguration sein können, da sie weder strukturellen noch funktionalen[63] Einfluss besitzen.

Die wichtigeren Unterscheidungsmerkmale liegen aber auf der Seite der Funktionalität eines Taschenrechners, beginnend etwa mit dem verwendeten Zahlensystem. Die Eingabe kann, wie dies meist der Fall ist, im Dezimalsystem erfolgen. Es wäre aber auch denkbar, dass mit hexadezimalen Zahlen gearbeitet werden soll. Die Wahl des Zahlensystems hat Auswirkungen auf mehrere Komponenten des Taschenrechners. Die Verwendung von hexadezimalen Zahlen ist nur sinnvoll, wenn es auch einen Eingabeblock gibt, der es ermöglicht Buchstaben wie A und B usw. einzugeben. Die Komponenten, die die eigentliche Berechnung durchführen, können möglichwerweise nur mit Zahlen im Dezimalformat umgehen. Es wird dann ein Konverter benötigt, der die Zahlen zwischen der internen und der externen Darstellung umwandelt. Theoretisch wäre es natürlich wünschenswert, die Möglichkeit zu haben, zwischen verschiedenen Zahlensystemen hin- und herzuschalten. Der enstehende Taschenrechner soll jedoch kein Universalrechner sein, wie man ihn aus der Praxis kennt, sondern ein für eine bestimmte Aufgabe optimierter Taschenrechner sein.

Ein Taschenrechner kann sich auch in der verwendeten Eingabenotation von anderen unterscheiden, etwa die Infix- oder die umgekehrt polnische Notation (RPN). Dies betrifft dann die Reihenfolge der Dateneingabe. Bei der polnischen Variante werden zuerst die Zahlen und dann erst der gewünschten Operator eingegeben, was zum Beispiel Vorteile bei der Eingabe von komple-

[63] im Sinne der Anwendungslogik

xen Brüchen u.ä. bietet, da keine Klammern eingesetzt werden müssen. Eine solche Notation hat dann auch Einfluss auf die benötigten Eingabekomponenten, etwa eine explizite Eingabetaste (`enter`), um das Ende einer Eingabe zu kennzeichnen.

Ein Taschenrechner kann auch eine beliebige Anzahl an Zwischenspeichern besitzen. Ferner benötigen einige Anwender neben den Grundrechenarten zusätzliche Funktionen. Hierbei kann es sich zum Beispiel um trigonometrische Funktionen wie Sinus, Cosinus oder Tangens handeln oder die Berechnung von Potenzen. Für einige Taschenrechner werden auch spezielle Funktionen benötigt, wie die Umrechnung von DM in Euro. Ein mit dem Generator produzierter Taschenrechner sollte eine beliebige Anzahl von Funktionen und Zwischenspeicher, die hierbei auch als funktionale Einheiten betrachtet werden können, erlauben, um eine entsprechende Produktvielfalt abdecken und individuelle »Kundenwünsche« berücksichtigen zu können.

5.12.2 Die Komponententypen eines Taschenrechners

Bei der Zerlegung eines Taschenrechners in einzelne Komponenten wird berücksichtigt, dass die einzelnen Komponenten gemäß des Komponentenparadigmas möglichst wenig Wissen voneinander haben sollten. Einige Komponententypen ergeben sich schon aus der Betrachtung eines »normalen« (physischen) Taschenrechners. Auch bei diesem existieren Einzelbausteine, wie Eingabetasten, ein Display und ein Gehäuse. Diese Bausteine werden direkt als Softwarekomponenten im Sinne einer interaktiven Benutzeroberfläche (GUI) übernommen. Schwieriger ist es, die interne Funktionsweise auf der Grundlage austauschbarer Einzelbausteine zu modellieren.

Ein Problem besteht etwa bei der Eingabe und Berechnung von Zahlen. Wenn als letztes eine Zahl berechnet wurde, beginnt bei einer darauffolgenden Zifferneingabe eine neue Zahl. Wenn zuletzt eine Zahl eingegeben wurde, ist die Eingabe der Zahl noch nicht beendet und die eingegebene Ziffer wird hinten an die bisherige Zahl angehängt. Es ist die Aufgabe einer Komponente, die hier als `Aktiver_Speicher` bezeichnet wird, zwischen diesen beiden Modi zu unterscheiden und die Zahlen an alle beteilgten Komponenten zu verteilen.

Ein ähnliches Problem betrifft die Rechenregeln. Ein erster Ansatz könnte darin bestehen, die Funktionalität hinter jede Taste als eigene Komponente abzubilden, das hieße etwa die Addition genauso zu behandeln wie eine trigonometrische Funktion. Es

wäre in diesem Fall notwendig zusätzliche Mechanismen einzubauen, die mathmatische Regeln wie Punkt- vor Strichrechnung berücksichtigen, wobei die Komponenten kein Wissen von anderen Komponenten der Anwendung haben. Zur Vereinfachung des Beispiels wurde hier stattdessen die Funktionalität, die Grundrechenarten bereitzustellen, auf eine einzelne (aggregierte) Komponente übertragen. Diese Komponente beinhaltet dann alle Funktionalitäten, die eine sehr enge Verknüpfung zueinander besitzen und stellt somit auch im Sinne der Granularitätsfrage eine sinnvollere Komponente dar. Zu ihr gehört auch eine Löschtaste (`Clear`-Button), der die aktuelle Eingabe und gleichzeitig auch alle Zwischenberechnungen löscht. Wenn die Komponente nur eine Zahl `0` erhält, würde sie nicht wissen, ob die Zwischenberechnungen auch zu löschen sind. Eine andere Möglichkeit wäre eine zusätzliche Nachricht, die alle Funktionseinheiten (s.u.) verstehen müssten.

Die zusätzliche Bereitstellung von trigonometrischen Funktionen oder ähnlichem stellt dagegen kein Problem dar, da diese nur die aktuelle Zahl verändern und keine zusätzlichen Auswirkungen auf andere Berechnungen haben. Komponenten, die Grundrechenarten implementieren, unterscheiden sich nur in ihrer Grösse (dem Umfang der von ihnen abgedeckten Funktionalität), aber nicht im eigentlichen Charakter von anderen Funktionskomponenten. Für alle wurde daher der gleiche Komponententyp (`Funktionseinheit`) gewählt. Es ist genauso möglich mehrere trigonometrische Funktionen in einer Komponente anzubieten. Die gewählte Modellierung hat den Vorteil, einen relativ einfachen Aufbau der Anwendung zu erlauben und das notwendige gemeinsame Verständnis von auszutauschenden Nachrichten zwischen Komponenten zu minimieren.

Die Tabelle 5.1 fasst in alphabetischer Folge die Komponententypen, ihre Funktion und Unterscheidungskriterien zusammen.

5.12.3 Interaktionsmuster der Taschenrechner-Domäne

Tabelle 5.2 gibt eine Übersicht über die verwendeten Nachrichten in Form von Ereignissen, ihre Bedeutung und welche Komponententypen sie als Ein- bzw. Ausgabe anbieten.

Die Funktionseinheiten wissen zu jedem Zeitpunkt, welche Zahl aktuell ist, d.h. auch in der Ausgabe (Display) für den Anwender zu sehen ist, wobei die Darstellung in einem anderen Zahlenformat vorliegen kann. Wenn eine Aktiviät einer Funkti-

Komponententyp	Beschreibung	Merkmale
Aktiver_Speicher	die Leitzentrale des Taschenrechners. Er weiss, ob eine Eingabe eine Zahl fortsetzt oder eine neue Zahl beginnt	keine
Button	stellt einen Button dar, der eine Aktion z.B. bei einer Funktionseinheit auslöst. Eine mögliche Implementierung ist etwa die Kapselung der Javaklasse *JButton* durch eine Schnittstellenbeschreibung.	Farbe, Form
Funktionseinheit	erledigt die eigentliche Rechnenarbeit. Es kann sich um eine einzelne Funktion handeln, die durch einen Taste aufgerufen wird oder um komplexere Einheiten, die z.B. die Grundrechenarten anbieten. Eine Funktionseinheit implementiert nur den logischen Teil. Die einzelnen Funktionalitäten müssen durch Tasten aufgerufen werden.	Funktion, Notation...
Gehaeuse	liefert den visullen Rahmen des Taschenrechners. Dieser Komponententyp bietet Methoden an, um Display, Eingabemöglichkeit und Buttons anzuzeigen	Farbe, Stil...
Input	ermöglicht die Eingabe von Zahlen. Die Eingabemöglichkeiten wurden so implementiert, dass eine Komponente selbst die grafische Eingabemöglichkeit bereitstellt und nicht auf andere andere Hilfsklassen wie Buttons zurückgreift. Dies dient hauptsächlich einer einfacheren grafischen Darstellung des Taschenrechners und entspricht dem Komponentendenken	Zahlensystem
Konverter	konvertiert Zahlen von einem Zahlensystem ins Dezimalsystem und umgekehrt	Zahlensystem
Output	gibt das Ergebnis der Berechnung oder der Eingabe aus.	Zeilen, Art

Tabelle 5.1
Komponententypen des Taschenrechners

onseinheit gestartet wurde, d.h. eine Zahl berechnet oder für eine Berechnung zwischengespeichert wurde, wird eine Nachricht versendet, um anderen Komponenten die Möglickeit zu geben,

auf dieses Ereignis zu reagieren bzw. sich auf die neue Situation einzustellen. Dies muss auch geschehen, wenn sich die aktuelle Zahl durch die entsprechende Aktivität nicht verändert, damit der `Aktive_Speicher` weiss, dass bei einer erneuten Ziffereingabe eine neue Zahl beginnt.

Name	**Beschreibung**	**Import**	**Export**
Action	löst eine Aktion aus	Funktionseinheit	Button
NumberCalculated	eine Zahl wurde berechnet oder für eine Berechnung verwendet	Aktiver_Speicher, Konverter	Funktionseinheit
NumberChanged	eine neue Zahl wurde eingegeben oder eine berechnete Zahl wird an alle weitergegeben	Funktionseinheit, Konverter, Display	Aktiver_Speicher, Konverter
NumberInput	eine Ziffer wurde eingegeben	Aktiver_Speicher	Input

Tabelle 5.2
Nachrichten der Interaktionsmuster der Domäne Taschenrechner

5.12.4 Anwendungsstruktur

Die identifizierten Komponententypen werden mit Hilfe der für sie festgelegten Nachrichten miteineinander in Beziehung gesetzt. Die resultierende Anwendungsstruktur wird in Form einer DTD-Datei repräsentiert und beinhaltet verschiedene Variationsmöglichkeiten. 5.11 zeigt einen möglichen Taschenrechner als Baumstruktur. Unterscheidungskriterien möglicher Taschenrechner werden durch die Verwendung unterschiedlicher Komponenten eines Typs (oder dessen optionales bzw. mehrfaches Auftreten) erreicht. Die Struktur selbst bleibt gleich. Es ist möglich, beliebig viele Funktionseinheiten einzusetzen. Alle anderen Komponententypen kommen genau einmal vor. Nur der Einsatz eines Konverters ist abhängig von der Auswahl der Zahlensysteme bei Ein-, Ausgabe und Funktionseinheiten, wobei letztere immer im dezimalen Zahlensystem arbeiten. Wenn die gleichen Zahlensysteme verwendet werden, ist kein Konverter notwendig und sollte dann auch vom Generator aus Optimierungsgründen nicht eingesetzt werden. Vergleiche auch die *Zero-Overhead-Rule* der generativen Programmierung in Abschnitt 5.6.

Das `Gehaeuse` des Taschenrechners ist von den eigentlichen Berechnungen vollständig unabhängig. Der Komponententyp, der mit der grössten Anzahl anderer Komponententypen in direkter Beziehung steht, ist der `Aktive_Speicher`. Er sollte

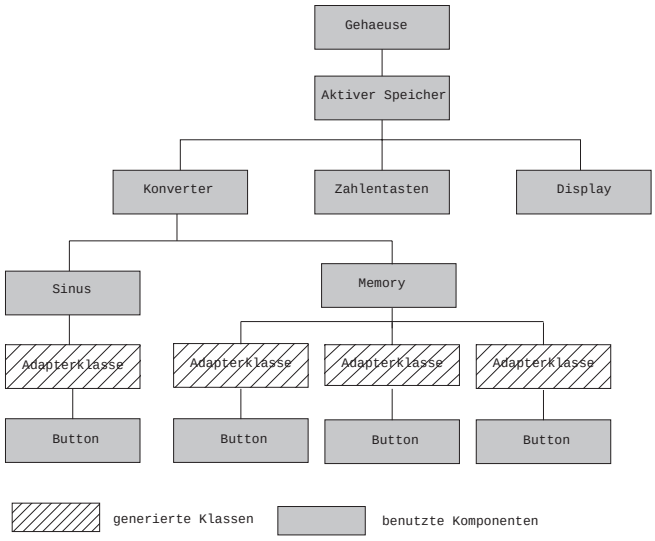


Abbildung 5.11 *Baumstruktur eines konkreten Taschenrechners*

daher in der Hierachie sehr weit oben stehen. Er muss die aktuelle Ziffer von der Eingabe annehmen und an die Anzeige (`Output`) weitergeben. Diese wird direkt mit `Aktiver_Speicher` verknüpft. Ausserdem muss die gesamte Zahl an die beliebig vielen Funktionseinheiten weitergegeben werden können. Dies geschieht falls notwendig über einen Konverter, der die konvertierte Zahl an die Funktionseinheiten weiterreichen kann. Damit ist die benötigte Anwendungsstruktur bereits weitgehend festgelegt. Abschnitt 5.12.6 diskutiert die speziellen Probleme *optionaler* Komponenten wie dem Konverter. Das Listing 5.1 zeigt die DTD, die als Architekturstil die Menge der möglichen Anwendungsstrukturen für die Domäne »Taschenrechner« festlegt und in dieser Form später vom Generator eingelesen wird, wenn ein Entwickler die entsprechende Produktlinie auswählt.

Listing 5.1*: Architekturstil eines Taschenrechners als DTD*

```
<?xml encoding="US-ASCII"?>
<!-- Taschenrechner Spezifikation V 1.0 -->
<!ELEMENT taschenrechner (gehaeuse)>
<!ELEMENT gehaeuse (aktiver_speicher)>
<!ELEMENT aktiver_speicher (input,output,konverter)>
<!ELEMENT input (#PCDATA)>
<!ELEMENT output (#PCDATA)>
<!ELEMENT konverter (funktionseinheit*)>
<!ELEMENT funktionseinheit (button)>
<!ELEMENT button (#PCDATA)>
```

5.12.5 Abhängigkeiten und Prioritäten

Die Komponenten des Taschenrechners können relativ unabhängig voneinander ausgewählt werden. Es wären aber zum Beispiel stilistische Merkmale bei der Visualiserung der Anwendung möglich, die mehrere Komponententypen betreffen. Das Aussehen von Tasten, Anzeige, Zahleneingabe und Gehäuse sollte miteinander harmonisieren. Die Merkmale hierfür wurden allerdings noch nicht im Rahmen der Domänenmodellierung festgelegt. Ein Merkmal, das mehrere Komponententypen betrifft, ist das Zahlensystem für Ein- und Ausgabe. Die Funktionseinheiten arbeiten allgemein im dezimalen Zahlensystem. Wenn die Zahlen zum Beispiel im hexadezimalen Zahlensystem eingegeben werden, muss, wie beschrieben ein Konverter die Umformung übernehmen. Ausserdem muss die Möglichkeit gegeben sein, hexadezimale Zahlen einzugeben.

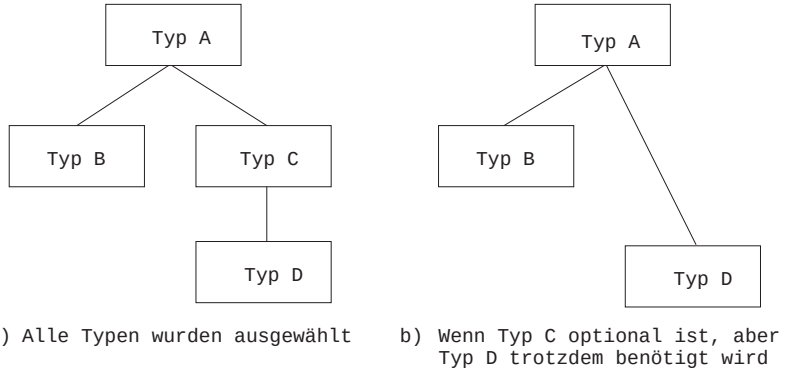
Daher sollte der spätere Dialog mit dem Konstrukteur mit der Abfrage des optionalen Komponententyps `Konverter` (s.a. folgenden Abschnitt) beginnen. Die dabei ausgewählten Merkmale, wie *Externes_Zahlensystem*, werden bei der Auswahl der restlichen Komponenten berücksichtigt. Die spezielle Bezeichnung wurde bei der Domänenanalyse für das extern, also von Eingabe und Ausgabe, verwendetete Zahlensystem eingeführt. Wenn für das *Externe_Zahlensystem* beim Konverter *hexadezimal* ausgewählt wurde, muss auch die Eingabemöglichkeit diese Eigenschaft besitzen. So wird sichergestellt, dass der Ziffernblock auch die Zahlen / Buchstaben `A` bis `F` anbietet, bzw. nicht anbietet, wenn *dezimal* als Merkmalswert ausgewählt wurde.

5.12.6 Behandlung optionaler Komponenten

Eine DTD erlaubt die Definition von optionalen Elementen, d.h. Elemente die vorkommen können aber nicht müssen, so wie es auch eine Architekturmodellierung mittels Merkmalsdiagrammen vorsieht (vgl. Abschnitt 6.4.3). Obwohl es sich bei einem Konverter, wie gesehen, prinzipiell um ein solches Element handelt, reicht es nicht aus den Komponententyp `Konverter` als optionales DTD-Element zu definieren, da die Elemente `Funktionseinheit` statt mit einem `Konverter` bei dessen Fehlen mit einer Komponente vom Typ `Aktiver_Speicher` verknüpft werden müssen. Es treten keine Probleme bei optionalen Elementen auf, wenn das Element selbst ein Blatt der betrachteten XML-Baumstruktur istalso keine Kindelemente besitzt, oder der Teilbaum unter dem Element ebenfalls als optional wegfällt.

In der exemplarischen Taschenrechner-DTD wurde festgelegt, dass das XML-Element `Konverter` mehrere Elemente vom Typ `Funktionseinheit`, d.h. einen ganzen Teilbaum, beinhalten kann. Wenn der `Konverter` optional wäre, würde ein gültiges XML-Dokument ohne `Konverter` auch keine Elemente vom Typ *Funktionseinheit* beinhalten. Daher werden Komponententypen, die weggelassen werden können, deren »Unterelemente« aber weiterhin Teil der Anwendung werden sollen, durch *abstrakte Typen bzw. Schnittstellen*, also Schnittstellen hinter denen sich keine konkrete Implementierung verbirgt, realisiert. Diese abstrakten Typen beinhalten die Merkmale, die auftreten, wenn keine Komponente von einem optionalen Typ benötigt wird, im Beispiel des Konverters also das Merkmal *Externes_Zahlensystem=dezimal*. Der Anwender kann dieses Merkmal genauso auswählen wie Merkmale von anderen Komponenten, sodass sich die erstellte Anwendung korrekt und wie erwartet verhält. Werden die Merkmale eines abstrakten Typs ausgewählt, wird an Stelle einer Komponente nur der unter ihr liegende Teilbaum mit dem Vaterknoten der Struktur verknüpft (Für diese Verknüpfung ist der `AppBuilder` zuständig, vgl. Abschnitt 6.3). Für den Konstrukteur bleibt es transparent, ob es sich um Merkmale konkreter Komponenten oder um Merkmale abstrakter Typen handelt, die zur Auswahl stehen.

Abbildung 5.12
a) Alle Typen wurden ausgewählt b) Wenn Typ C optional ist, aber Typ D trotzdem benötigt wird



Der Baum in Abbildung 5.12 a) illustriert nochmals die Problemtik. Typ B und D könnten ohne Probleme weggelassen werden, wenn sie als optional definiert sind. Anders sieht es aus, wenn Typ C optional ist. Es kann zum einen notwendig sein, auch Typ D, d.h. den gesamten Teilbaum unter Typ C wegzulassen. Auch hierbei treten keine weiteren Probleme für den Generator auf. Anders sieht es aus, wenn Typ D weiterhin Teil der

Anwendung sein soll, auch wenn Typ C optional ist und nicht gewählt wurde (s. Abbildung 5.12 b) und Beispiel Konverter). Dies wird durch die Einführung der oben beschriebenen abstrakten Typen kompensiert. In der Beschreibung der DTD darf das entsprechende Element dabei nicht als optional spezifiziert sein. Bei der Zusammenstellung der Anwendung aus dem intern generierten XML-Dokument erkennt der Generator die leere Implementierung. Der vom Typmanagement verwalette *ComponentPool* selbst unterscheidet nicht zwischen Typen, für die Komponenten vorliegen oder abstrakten Typen. Der Generator verknüpft den Teilbaum unter diesem Komponententyp mit dem vorgesehenen Vaterknoten, im Beispiel Typ D mit Typ A (s. Abbildung 5.12 b)). Dabei muss der Vater (also A) die gleiche Schnittstelle zu D bereitstellen, wie es Typ C getan hätte, damit eine gleichbleibende Funktionalität gewährleistet bleibt. Es handelt sich also im Sinne von Abschnitt 5.7.1 um *symmetrische* Komponenten. Diese Eigenschaft ist für optionale Komponenten in der Regel ohnehin zu fordern, um ihr variables Auftreten überhaupt vorsehen zu können.

5.13 Zusammenfassung

5.13.1 Generative Softwarekonstruktion

Mit Hilfe des Komponentenparadigmas und den damit einhergehenden Abstraktionsmechanismen lässt sich die Vorstellung einer industriellen Softwareproduktion mit dem Ziel der Erstellung *kompositorischer* Anwendungssysteme auf der Basis vorgefertigter Softwarekomponenten neu beleuchten. Wesentlich ist dabei der Wunsch nach einer Produktivitäts- sowie Qualitätssteigerung in der Softwareentwicklung. Während durch die Objektorientierung zwar eine Verbesserung in diesem Bereich erreicht wurde, so führt sie in der Praxis oft immer noch zu monolithischen, projektbezogenen Individuallösungen.

Die Idee der Generator-basierten Anwendungserstellung soll an dieser Stelle Abhilfe schaffen, indem sie von der programmiersprachlichen Ebene abstrahiert und die Anwendung als ein Gesamtsystem interagierender Komponenten betrachtet [Griffel, 2000a]. Der Softwaregenerator stützt sich dabei auf die Komposition (Zusammensetzung) *vorhandener* Softwarekomponenten. Durch eine Automatisierung dieses Prozesses können somit robuste, fehlerfreie Anwendungssysteme hoher Komplexität in kurzer Zeit erstellt werden. Der Begriff der *Softwarekonstruktion* spiegelt insbesondere eine klare Abgrenzung zum Begriff der

Programmierung wider, sodass bei diesem Vorgehen keine Programmierkenntnisse notwendig sind.

5.13.2 Generierungsphasen

Die generative Konstruktion eines Anwendungssystems basiert auf der oben gezeigten Grundkonzeption, die sich in drei wesentliche Phasen unterteilen lässt.

In der ersten Phase – der *Spezifikationsphase* – wird der Generator mit Hilfe einer Architekturspezifikation – dem Anwendungsbauplan – parametrisiert. Die Architekturspezifikation basiert dabei auf einer domänenspezifischen »Eingabesprache«, gegebenenfalls kombiniert mit einer »Konfigurationssprache«, die aus der Analyse der zu Grunde liegenden Domäne gewonnen werden kann. Somit kann der Generator nachfolgend passende Komponenten*typen* auswählen, deren Komponentenspezifikation eben gerade die Architekturspezifikation befriedigt. Ein entspechend geeignetes Typkonzept ist daher unabdingbar.

Im Anschluss betrachtet die *Konfigurationsphase* die Konfiguration der in der ersten Phase ausgewählten Komponenten. Der Generator stützt sich dabei auf sein Konfigurationswissen, das sich aus den Kompositionsregeln zusammensetzt, die die Randbedingungen der Konfigurationsmöglichkeiten ausdrücken – den gültigen Komponentenkombinationen. An dieser Stelle können somit Freiheitsgrade entstehen, die in Anwendungen mit einem hohen Maß an Variabilität resultieren. Diese werden dann in ihrer Gesamtheit als Produktlinien bezeichnet. Nach erfolgreichem Abschluss dieser (interaktiven) Konfiguration (engl. *Customizing*) kann gemäß der Architekturspezifikation ein Anwendungsskelett generiert werden, in dem alle Verknüpfungspunkte explizit definiert sind. Dabei wird abermals das Typwissen benötigt. Festzuhalten bleibt jedoch auch, dass der Zusammenhang zwischen Domänenanalyse und Architekturmodellierung auch heute noch nicht vollständig verstanden wird [Bass et al., 1998].

In der letzten Phase – der *Konstruktionsphase* – werden Instanzen der im Anwendungsskelett spezifizierten Komponententypen endgültig selektiert und tatsächlich (technisch) verknüpft, damit diese untereinander (auf technischer Ebene) kommunizieren können, wobei wiederum auf das Typwissen zurückgegriffen wird. Abgeschlossen wird die Generierung durch die Erstellung eines Anwendungsarchivs, das alle Bestandteile der konstruierten Gesamtanwendung beinhaltet (Komponentencode, Skelettcode, Deskriptoren etc.). Es repräsentiert zugleich eine ausführbare Anwendung einer Produktfamilie, die sich intern durch einen ho-

hen Wiederverwendungsgrad einzelner Komponenten auszeichnet.

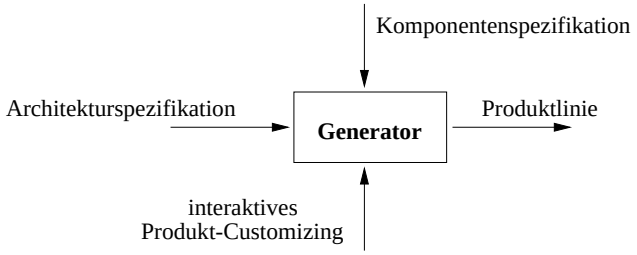


Abbildung 5.13 *Grundkonzeption einer generativen Systemerstellung*

5.13.3 Einordnung und Bewertung

Der vorgestellte Ansatz abstrahiert von konkreten Programmiersprachen. [Batory et al., 1994] vertreten die These, dass je eher Konstrukte eines Softwaremodells sich in einer Programmiersprache wiederfinden, sie um so leichter für den Entwickler verständlich sind.[64] Der Autor legt in dieser Arbeit einen andere Vorstellung von einem Entwickler zu Grunde, nämlich die des *Konstrukteurs*, der gerade nicht mit technischen Details belastet werden möchte. Daher werden direkt mit Implementationssprachen assoziierte Modelle für zu konkret (in einem zu frühen Stadium) gehalten und verlieren entsprechend ihren abstrahierenden Charakter.

Werden als Ausgangspunkt bestehende Komponenten gewählt, ist eine stärkere Trennung zwischen der Systemmodellierung und dem Übergang zu konkreten Techniken gegeben, die eine jeweilige Konzentration auf beide Teilbereiche erlaubt. Die *Abstraktion* (Struktur, Datentypen, Algorithmen etc.) ist dabei vollständig, da auf fertige Komponenten zurück gegriffen wird. Um dennoch den gewünschten bruchlosen Übergang unter Aufrechterhaltung der im Modell zugesicherten Eigenschaften in eine konkrete Implementation zu erlauben, wird eine entsprechend aussagekräftige *deskriptive* Vorgehensweise in Form von Typ- bzw. Komponentendeskriptoren vorgeschlagen. Die die-

[64] Gleichzeitig führt die Sprachnähe solcher Ansätze häufig zu Problemen, siehe z.B. die Diskussion des Fehlens einer Typisierung verschachtelter Klassen in Java in [Batory & Smaragdakis, 2000]. Dies führt dann dazu, dass entsprechende Spracherweiterungen postuliert werden. Der Autor ist der Meinung, dass dieses babylonische Vorgehen in der Informatik nicht mehr zeitgemäß ist.

sen zu Grunde liegenden XML/DTD-Repräsentationen lassen eine fachbezogenere bzw. domänennähere Beschreibung etwa im Vergleich zu GenVocas Attributgrammatiken zu, ohne jedoch die für eine verifizierende Arbeitsweise des Generators zur Vermeidung semantisch inkorrekter Kompositionen notwendigen formalen Hilfsmittel (logische Richtlinien und Randbedingungen sowie Protokolle) zu opfern.

Schwerpunkt der Betrachtungen des generativen Prozesses sind dabei nicht die visuelle Führung durch den Generatorvorgang, obwohl explizit eine solche »Anleitung« des Konstrukteurs gefordert wird. Hierzu existieren eine Reihe von Arbeiten (s. etwa [Batory et al., 1999a]), obwohl für eine rein komponentennutzende Konstruktion bislang noch wenig Erfahrungen für eine optimale Entwicklerunterstützung vorliegen.[65] Den wesentlichsten Beitrag hierzu leistet die vorliegende Arbeit durch das im voranstehenden Kapitel beschriebene Typmanagement, das dem Anwendungskompositeur entspechende Informationen liefert.

Die eingesetzte DTD-Repräsentation *forciert* die Schichten-Bildung (engl. *Layer*) für eine Anwendung, während programmiersprachliche Hilfsmittel wie etwa eine Modulbildung dies zwar erlauben, aber eben nicht erzwingen. Die Layer-Diskussion in [Batory & Smaragdakis, 2000] zeigt, dass die strenge der XML-Hierarchie kein Problem ist, sondern einzelne Ebenen eben Mengen (hier für Architekturmerkmale) austauschbarer Elemente darstellen. Die Kombination von Elementen verschiedener Ebenen entspricht dann der Bildung von Vererbungshierarchien, wobei dieser Aspekt hier bewusst ausgegrenzt wird, da lose gekoppelte Anwendungsstrukturen gefordert werden und nicht auf Sprachebene argumentiert werden soll. Nach [Batory & Smaragdakis, 2000] wäre dies – konform mit der hier vorgenommenen Begriffsbildung – als Kollaboration zu bezeichnen.

Die Spezifikationsphase des generativen Prozesses ist trotz einer vorliegenden Architekturbeschreibung notwendig, da das »Passen« einer isoliert betrachteten Komponente nicht notwendigerweise auch zu passenden Komponenten in der Kombination mit weiteren führt. Konfigurationsentscheidungen zu einzelnen Komponenten legen zum Beispiel eine differierende Auswahl aus einer Menge von »zulässigen« Komponenten nahe. Die Domänenanalyse beugt hier jedoch in gewissen Maße vor, indem sie

[65] Grafische Oberflächen wie Batory sie in [Batory et al., 1999a] beschreibt, entsprechen eher Property Editoren [Vanhelsuwé, 1997] und lösen damit nicht das Problem eine aus den Komponentenspezifikationen *abgeleitete* Oberfläche zu bieten.

ja zunächst Merkmale eines Anwendungsbereichs im gegenseitigen Verhältnis (also nicht-isoliert) betrachtet und erst im Anschluss eine Abbildung auf Komponenten erfolgt. Ebenso trägt eine explizite Deployment-Information, wie sie heute für Komponentenmodelle vorgesehen ist, dazu bei, die korrekte Funktionsweise in der endgültigen Einsatz/Laufzeit-Umgebung sicherzustellen [Broschei, 2000].

Die Rückführung der vom Generator erstellten XML-Deskriptoren der generierten Applikationskomponente als Typbeschreibung in den Typmanager erlaubt auch Tests nach außen bzw. gegeneinander (einzelner Komponenten). Technisch ist damit zwar keine Zusammenführung von Schnittstellen(methoden) realisierbar (vgl. aber etwa [Moehl, 1999b] und [Martens, 1999]), aber auf Modellebene erlaubt dies entsprechende Aggregationstests. Im Vergleich zu Allens Ansatz [Allen, 1997] steht allerdings eine korrekte *Konstruktion* statt Tests und Analysemöglichkeiten im Nachinein im Vordergrund. [Allen, 1997] bietet eine gute Übersicht, welche Arten von Konsistenzprüfungen unterschieden bzw. welche Konstrukte einer solchen Prüfung unterzogen werden können.

Dabei ist auch das Problem eines »*architectural mismatch* im Sinne von Garlan [Garlan et al., 1995a] zu berücksichtigen, da einzelne vorgefertigte Komponenten meist bestimmten Architekturstilen folgen (vgl. Shaw in [Shaw, 1995]). Das vorgestellte Typmodell versucht diesen Anforderungen durch die Modell- und Kommunikationstypen der syntaktischen 3. Ebene gerecht zu werden (vgl. Kapitel 4), die eine Abbildung (engl. *matching*) solcher technisch orientierten Architekturstile erlaubt.

Im Vergleich zu dem bislang in der Theorie erfolgreichsten generativen und domänenzentrierten Modell GenVoca, kann zunächst eine Analogie zum Begriff des Realm gezogen werden: Dies entspricht in der Terminologie der vorliegenden Arbeit einer Menge typkonformer Komponenten. Daher ist die Eigenschaft der GenVoca-Komponenten innerhalb eines Realm »plug-compatible« und »interchangeable« nicht überraschend. Dies löst jedoch nicht das Problem der Kompatibilität von Komponenten, da das der Frage nach dem Passen zweier Komponenten aus zwei Realms entspräche. Realms als Grammatiken, deren Sätze dann »Software« sind, ist eine elegante Idee, aber GenVoca umgeht die Frage der Typisierung auch insofern, als dass von standardisierten Einheitsschnittstellen zur einfachen Realisierung der Kompositionsfähigkeit ausgegangen wird [Batory & Geraci,]. Fraglich ist, ob sich die Softwareindustrie jemals auf solche Standards wird einigen können. Der Autor hält daher »abbildende«

(engl. *mapping*) Ansätze für erfolgversprechender (im Sinne der Praxisakzeptanz und damit letztlich dem Einsatzerfolg).

Die in dieser Arbeit eingenommene interaktionszentrierte bzw. konnektionistische Sicht entspricht in der Terminologie von Batory einem »*architectural style programming*« [Batory et al., 1999b]. Er bezeichnet hiermit die Trennung der Funktionalität einer Komponente von deren Art der Kommunikation mit anderen Komponenten im Rahmen einer Anwendung. Eine solche Trennung erlaubt dann auch die direkte Platzierung von Adaptern (vgl. a. [Griffel, 1996] und Kapitel 3).

5.13.4 Konzeption einer vollständigen Entwicklungsumgebung

Die Sinnhaftigkeit und Notwendigkeit eines expliziten Typmanagements für eine generative Entwicklungsumgebung ist deutlich geworden. Ein solcher *Unterstützungsdienst* kann von einer Typmanagement-Komponente erbracht werden, die sich auf die Evaluierung von Beziehungen zwischen Komponenten auf Basis ihrer Typbeschreibungen sowie die Ableitung von Aussagen über Komponententypen gemäß des Modells aus Kapitel 4 stützt und dadurch einen obligatorischen Bestandteil der generativen Entwicklungsumgebung ausmacht [Griffel, 2000a].

Der Typmanager nimmt in diesem Kontext für alle Komponenten die Rolle einer zentralen Registrierungsinstanz ein, wobei die durch ein Modellierungswerkzeug – den Merkmalsdiagramm-Editor, vgl. Abschnitt 6.4.3 – gegebenen Meta-Informationen in Form von Merkmalsstrukturen (Feature Structures) [Carpenter, 1992], [Wienberg, 2001] in den Aufbau der Komponentenspezifikation eingehen, die nachfolgend den Komponententyp repräsentiert. Die Spezifikation des Meta-Modells in Form einer Typbeschreibung für die einzelnen Komponenten muss für die einzelnen Generierungsphasen Aussagen über die *Konformität* zur Architektur und zur *Kompatibilität* der Kompositionen erlauben.

Konformitäts- wie auch Kompatibilitätsbeziehungen, die typischerweise zwischen Komponenteninstanzen zum Tragen kommen, können in diesem Zusammenhang vielmehr auch als Relationen zwischen Komponententypen aufgefasst werden, da sich derartige Beziehungen stets auf Basis der Typbeschreibung von Komponenten ergeben, die ja gerade den Komponententyp repräsentiert. Sie lassen sich daraufhin auf die zugeordneten Komponenteninstanzen übertragen.

Neben der Komponenten- sowie Architekturspezifikation erhält der Generator Zugriff auf die Komponenteninstanzen des Komponenten-Repository. Letztendlich erlaubt ein Customizer-Wizard eine interaktive Produktkonfiguration, die dem Generator als Eingabe zur Verfügung steht, sodass dieser daraufhin die Generierung einer Applikation einer einzelnen Produktlinie vornehmen kann.

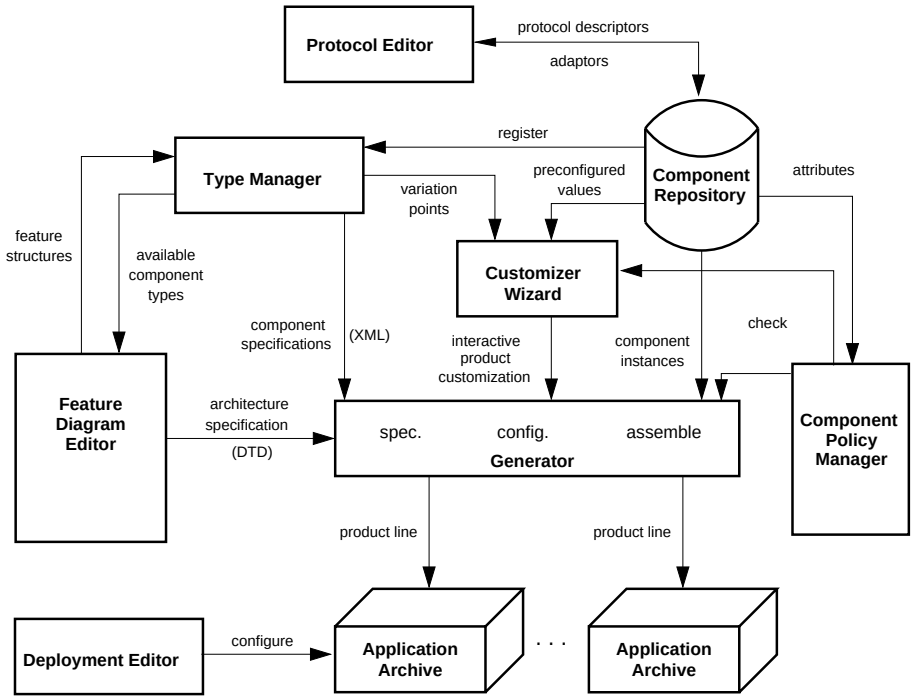


Abbildung 5.14
Prinzipsicht der generativen Entwicklungsumgebung

Wird das Gesamtkonzept in Form einer solchen kompletten Systemumgebung betrachtet, kann als früher Vorläufer dieser Ideen zum Beispiel das 001-System [Krut, Jr., 1993] betrachtet werden. Eine entsprechende Zusammenführung aller bislang vorgestellten Konzepte im Rahmen ihrer konkreten Implementation sowie die Ergänzung um entsprechende Werkzeuge zur Unterstützung des Konstruktionsprozesses ist nun Gegenstand des folgenden Kapitels, mit dem Ziel des Nachweises der Praktikabilität der vollständigen Prinzipsicht gemäß Abbildung 5.14.

6 Implementation

Die folgenden Abschnitte erörtern und illustrieren relevante Details der im Rahmen dieser Arbeit durchgeführten prototypischen Implementation der in den vorangegangenen Kapiteln diskutierten Konzepte und Mechanismen mit dem Ziel der Bereitstellung einer vollständigen Entwicklungsumgebung zur Unterstützung einer generativen Anwendungskonstruktion und deren Praktikabilititätsnachweis. Die Implementationen wurden auf Basis des Java Development Kit (JDK) 1.2.2 [Sun, 1999a] vorgenommen.

6.1 Die Adapter

Zunächst soll die konkrete Umsetzung der in Kapitel 3 erörterten Adaptermechanismen dargestellt werden. Die entwickelten Klassen, die ein prototypisches Rahmenwerk für Komponenten mit entsprechend erweiterten Schnittstellenspezifikationen darstellen, werden auszugsweise präsentiert und erläutert. Auf technische Probleme der Implementierung wird dabei ebenso eingegangen, wie auf die besonderen technischen Anforderungen an potenzielle Komponentenentwickler, deren Komponenten den vorgestellen Ansatz nutzen sollen.

Zur konkreten Anwendbarkeit der protokollbasierten Komponentenkommunikation und -adaption wurde eine Systemumgebung für Java-Objekte als exemplarische Komponenten entwickelt. Dieses soll die Realisierbarkeit der Implementierung eines derartigen Konzeptes aufzeigen und zugleich die Grundlage für die spätere Bereitstellung konkreter Komponenten zur Anwendungsgenerierung bilden. Konkret bietet diese Konnektorbasierte Systemumgebung folgende Dienste:

- Komponenten kommunizieren kontrolliert miteinander
- Kompatibilität zweier Komponenten kann formal geprüft werden

- Einbindung von Adaptoren zur Überwindung von Inkompatibilitäten ist möglich
- Adaptoren können halbautomatisch erzeugt werden

Zur die Ermöglichung dieser Dienste werden jedoch Anforderungen an den Komponentenentwickler gestellt.

6.1.1 Anforderungen an Komponentenentwickler

Das Konzept der im Folgenden präsentierten Systemumgebung erwartet von Komponentenentwicklern, deren Komponenten [1] in diese Umgebung eingebunden werden sollen, bestimmte Vorleistungen. Ein derart implementiertes Java-Objekt unterscheidet sich daher von gewöhnlichen Java-Objekten.[2] Konkret handelt es sich dabei um die Angabe einer Kollaborationsspezifikation und die Implementierung entfernter Methodenaufrufe mittels Ereignissen (engl. *Events*).

6.1.1.1 Kollaborationsspezifikation

Das Konzept und der Inhalt einer Kollaborationsspezifikation wurden in Abschnitt 3.3.1 erläutert. Konkret eingebunden werden diese Informationen vom Komponentenentwickler in mehrdimensionalen Feldern.[3]

[1] die sich aus einem oder mehreren Java-Objekten zusammensetzen

[2] In [Zirpins, 1999] wurde eine Bewertung der Java-Konzepte unter den Kriterien des Komponentenbegriffs vorgenommen und auf die Notwendigkeit entsprechender Erweiterungen hingewiesen, um zu einem Komponentenmodell gemäß des Kapitels 2 zu gelangen.

[3] Dieses implantierende Vorgehen entspricht zunächst dem Ansatz aus [Griffel et al., 1996] und wird später durch die Entnahme der zugehörigen Informationen aus den Deskriptoren des Typmodells ersetzt. Im Sinne des Komponentenparadigmas sollte jedoch auch ein Mechanismus zur Verfügung stehen, der die konnektionistische Sichtweise auch ohne explizites Typmanagement erlaubt. Der Protokolleditor (vgl. Abschnitt 6.4.2) generiert aus der hier gezeigten Syntax allerdings auch gleich die entsprechenden XML-Deskriptoren. In Anhang C findet sich die analoge Darstellung eines Bankszenarios wie es Gegenstand der Illustration des Typmanagements in Abschnitt 6.2 ist.

***Listing 6.1**: Implementierung der Kollaborationsspezifikation eines Auktionators nach Beispiel 3.1*

```
//Methodenaufrufe
String[][] receive = { {"bid", "bidderId:Id",
                        "itemBiddingOn:java.lang.String",
                        "amount:java.lang.Integer"},
                       {"updateAck"} };
String[][] send    = { {"newItem", "itemDescr:java.lang.String",
                        "bidderId:Id"},
                       {"update" , "highBid:java.lang.Integer"},
                       {"rejectBid", "reason:java.lang.String"},
                       {"acceptBid"},
                       {"itemSold"} };
//Protokoll
String      init    = "A";
String[]    finals  = {"E"};
String[]    trans   = { "A", "B", "-newItem",
                        "B", "C", "-update",
                        "B", "A", "-itemSold",
                        "B", "D", "+bid",
                        "C", "B", "+updateAck",
                        "D", "B", "-rejectBid",
                        "D", "E", "-acceptBid",
                        "E", "C", "-update"
                        "E", "A", "-itemSold"};
//Container erzeugen
MachineDescriber md = new MachineDescriber(init,finals,trans,receive,send);
//Spezifikation publizieren
public MachineDescriber getMachineDescriber() {
  return md;
}
...
```

Das zweidimensionale Feld `receive` listet die von dieser Komponente zu empfangenden Methodenaufrufe und die dazugehörigen Parameter auf. Dabei ist für jede Methode ein untergeordnetes Feld vorgesehen, welches als erstes Element den Methodennamen und als weitere Elemente Angaben zu den Parametern in der konkreten Form `name:typ` des Parameters enthält. Das zweidimensionale Feld `send` ist identisch aufgebaut und enthält die Methodenaufrufe, die von dieser Komponente ausgehen können.

Es folgt die Protokollspezifikation, die mit der Angabe von Anfangs- und Endzustand (potenziell sind mehrere Endzustände möglich, vgl. Kapitel 3) in `init` und `finals` eingeleitet wird. Das eigentliche Protokoll befindet sich im Feld `trans`, wobei die Zustandsübergänge in Gruppen von drei Elementen angeordnet sind. Dabei beinhaltet das erste Element den Ausgangszustand, das zweite Element den Zielzustand und das dritte den den Zustandsübergang auslösenden Methodenaufruf in der Form $\pm$`name` womit auch die *Richtung* des Methodenaufrufes festgelegt wird. Demnach beschreibt beispielsweise die erste Zeile in Listing 6.1 den Übergang $\mathtt{A} : +\mathtt{newItem} \mapsto \mathtt{B}$.

Die eigentliche Deklaration dieser Kollaborationsspezifikation erfolgt mittels der Erzeugung eines Objektes vom Typ `MachineDescriber`, das lediglich als ein Container für die übergebenen Variablen(felder) fungiert und hier nicht gezeigt ist. Der Zugriff auf diese Kollaborationsspezifikation wird nun über die Methode `getMachineDescriber()` ermöglicht.

6.1.1.2 Entfernter Methodenaufruf mittels `SignaturEvent`

Ein externer Methodenaufruf (eine zu sendende Nachricht) muss durch das Senden eines Ereignisses, das die zu sendende Nachricht transportiert, realisiert sein. Konkret handelt es sich dabei um ein Objekt vom Typ `SignaturEvent`.

***Listing 6.2**: Die Klasse* `SignaturEvent`

```
class SignaturEvent extends java.util.EventObject {

  public Object source;
  public String method;
  public int number_of_param;
  public Object[] param;
  public Class from; //Die Klasse zu "source"

  public SignaturEvent(Object s, String m, int n, Object[] p, Class f) {
    super(s);
    source = s;
    method = m;
    number_of_param = n;
    param = p;
    from = f;
  }
  ...
  //methoden zum auslesen
  ...
```

Dieses Ereignis beinhaltet Informationen über das aufrufende Objekt, den Methodennamen, die Anzahl der Parameter und die Parameter selbst (im Feld `param`). Für die Durchführung eines entfernten Methodenaufrufs muss somit ein `SignaturEvent`-Objekt mit den entsprechenden Daten erzeugt werden. Dieses wird dann mittels einer Methode den registrierten `EventListener`-Objekten für den `SignaturEvent` übergeben. Ein beispielhafter entfernter Methodenaufruf hat demnach folgendes Aussehen:

***Listing 6.3**: Exemplarischer entfernter Methodenaufruf*

```
...
//parameter verpacken
Object[] v = {itemDescr, id};
//Ereignis senden (feuern)
```

```
fireEvent( new SignaturEvent(this, "newItem", 2, v, this.getClass()));
...
```

Eine entsprechende `fireEvent`-Methode und Methoden zur Registrierung und Abmeldung von Objekten des Typs `SignaturListener` müssen entsprechend den Prinzipien der Ereignis-basierten Kommunikation in Java ebenfalls implementiert sein.

***Listing 6.4**: Das Interface* `SignaturListener`

```
public interface SignaturListener extends java.util.EventListener {
  public void transfer(SignaturEvent e);
}
```

6.1.1.3 Methodenaufruf auf eine Komponente

Im Gegensatz zu von einer Komponente ausgehenden Methodenaufrufen sind externe Methodenaufrufe auf eine Komponente (Nachrichtenempfang) für den Komponentenentwickler transparent. Er bietet die betreffende Methode durch eine Deklarierung als `public` anderen Komponenten zum Aufruf an – gemäß der vorgesehenen Java-Syntax.

6.1.2 Systemarchitektur: Kommunikationsschema

Die hier implementierte Systemarchitektur sieht vor, dass jegliche Kommunikation über Konnektoren (vermittelnde Objekte) erfolgt, die die Protokollkompatibilität prüfen und Methodenaufrufe an die jeweiligen Komponenten weiterleiten (siehe Abschnitt 3.3.2). Die Kopplung zweier Komponenten erfolgt daher durch die Kopplung von zugehörigen `Extension`-Objekten, die den Konnektor realisieren und miteinander mittels `SignaturEvent`-Objekten kommunizieren. Die Kommunikation zwischen `Extension`-Objekt und zugehöriger Komponente ist asymmetrisch: Die Kommunikation von Komponente zu Vermittler-Objekt geschieht mittels `SignaturEvent`; die Kommunikation von Vermittler-Objekt zu Komponente geschieht mittels eines direkten Methodenaufrufs auf die Komponente.

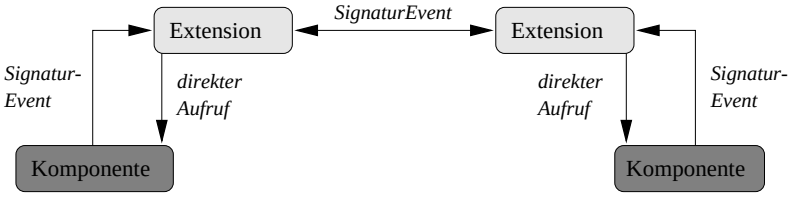
6.1.2.1 `Extension`-Objekte

Es wird für jede Komponente, die in das System eingegliedert werden soll, ein `Extension`-Objekt erzeugt, das bei seiner Initialisierung

- die Kollaborationsspezifikation einliest,
- aufgrund des Protokolls einen endlichen Automaten erzeugt und diesen initialisiert,
- sich bei der zugehörigen Komponente als `SignaturListener` (`Extension`-Objekte implementieren dieses Interface) registriert und
- die seitens der Komponente angebotenen Methodenaufrufe per Introspektion einliest.[4]

Die Erzeugung (und die später benötigte Weiterschaltung) des Automaten erfolgt mittels eines von [Merz, 1997] entwickelten und im Rahmen dieser Arbeit durch [Moehl, 1999a] erweiterten Frameworks, das Mechanismen zur Erzeugung, Prüfung, Weiterschaltung und der Abfrage nach möglichen Zustandsübergängen von Automaten bereitstellt, basierend auf unterschiedlichen Eingabeformaten, die auf Grund unterschiedlicher Syntax beispielsweise der Protokoll- bzw. Adapterprotokollspezifikation benötigt werden.

Abbildung 6.1 *Nachrichtenaustausch mittels* `Extension`*-Objekten*



Die eingelesenen Methodensignaturen werden für die im Verlauf der Kommunikation auftretenden direkten Methodenaufrufe auf die Komponente benötigt.

Nach Abschluss der Initialisierung kommt `Extension`-Objekten die Aufgabe der Weiterleitung von Nachrichten bei gleichzeitiger Prüfung auf Konformität mit dem in der Kollaborationsspezifikation angegebenem Protokoll zu. Diese Aufgabe wird von der zur Implementierung von `SignaturListener` benötigten Methode `transfer` übernommen.

[4] Dieses Vorgehen ist natürlich nur in reflektiven Sprachen möglich. Der verallgemeinernde Ansatz ist daher die Entnahme dieser Schnittstelleninformationen aus einem sprachneutralen XML-Deskriptor.

***Listing 6.5**: Die Klasse* `Extension`

```
public class Extension implements SignaturListener{

  MachineDescription target;
  Class targetclass;
  MachineDescriber md;
  Simulator s;
  String[] transmessages;
  Method[] targetmethods;
  private SignaturListener listener;
  //Initialisierung
  public Extension (MachineDescription target) {
    int count = 0;
    String[] trans;
    this.target = target;
    //Kollaborationsspezifikation einlesen
    md = target.getMachineDescriber();
    //Protokoll einlesen
    trans = md.getTrans();
    transmessages = new String[trans.length];
    while (count < trans.length) {
      count += 2;
      transmessages[count/3] = trans[count];
      count++;
    }
    s = new Simulator();
    //Automaten initialisieren
    s.buildMachine(md.getInit(), md.getFinals(), trans);
    //bei Komponente registrieren
    target.addSignaturListener(this);
    //methoden auslesen
    targetclass = target.getClass();
    targetmethods = targetclass.getDeclaredMethods();
  }
  //Methode fuer Nachrichtenweiterleitung
  public void transfer(SignaturEvent e) {
    String method = e.method;
    boolean stopped;
    String methodname = method.substring(1);
    SignaturEvent e_new = new SignaturEvent(e.source, e.method,
                                            e.number_of_param, e.param);
    if (method.charAt(0) == '-') { //kommunikationsrichtung?
      //entfernter Aufruf
      stopped = s.step(method);
      if (stopped) { //abbrechen
        System.out.println("Kommunikation beendet");
        System.exit(0);
      }
      else {
        e_new.method = "+"+methodname;
        fireEvent(e_new);
        return;
      }
    }
    else { //lokaler Aufruf
      stopped = s.step(method);
      if (stopped) { //abbrechen
```

```
      System.out.println("Kommunikation beendet");
      System.exit(0);
    }
    else {
      for (int i = 0; i < targetmethods.length; i++) {
        String name = targetmethods[i].getName();
        if (name.equals(methodname)) {
          try {
            targetmethods[i].invoke(target, e.param);
          }
...
return;}}}}}
```

`Extension`-Objekte dienen nicht nur dem Kommunikationsablauf der Komponentenkollaboration als kontrollierende Interzeptoren, sondern speichern auch die für die in den Abschnitten 6.1.3 und 6.1.7 vorgestellten Konzepte der Kompatibilitätsprüfung und halbautomatischen Adaptererzeugung benötigten Daten und bieten den für die zugehörigen Algorithmen notwendigen Zugriff auf die entsprechenden endlichen Automaten.

6.1.2.2 Kopplungsvorgang

Das Herstellen einer Verbindung zweier Komponenten mittels ihrer zugehörigen `Extension`-Objekte (durch Registrierung als `SignaturListener` auf beiden Seiten der Verbindung) ist Aufgabe einer übergeordneten Instanz – in diesem Falle später des Typmanagers und des Generators, die dann auch den Aufruf der Kompatibilitätsprüfung, das Einbinden eines Adapterprotokolls und den Aufruf der halbautomatischen Erzeugung eines Adapters anstoßen.

6.1.2.3 Synchronisation zur Vermeidung von Laufzeitproblemen

Um Laufzeitprobleme (vgl. Abschnitt 3.5) zu vermeiden, lässt sich die bislang beschriebene Architektur dahingehend erweitern, zu ermöglichen, dass vor dem Senden einer Nachricht zunächst eine Anfrage an einen zwischen die beteiligten `Extension`-Objekte eingefügten Vermittler gestellt werden kann, ob das Senden einer bestimmten Nachricht in diesem Zustand legal ist. Eine solche Anfrage könnte beispielsweise wiederum durch ein Ereignis realisiert sein. Vom Vermittler wird eine Erlaubnis zum eigentlichen Senden der Nachricht erteilt oder dieses abgelehnt. Im weiteren zeitlichen Ablauf erwartet der Vermittler das Senden dieser Nachricht und wird entsprechend andere Kommunikationswünsche ablehnen. Ein derartiges Konzept illustriert Abbildung 6.2. Die Rolle der Vermittlers kann der Typ-

manager übernehmen. Dies entspricht einer Laufzeiteinbindung des Typmanagements, sodass entsprechende Konsistenztest beispielsweise im Falle einer Laufzeitaktualisierung (Update) einer Applikationskomponente durchgeführt werden können.

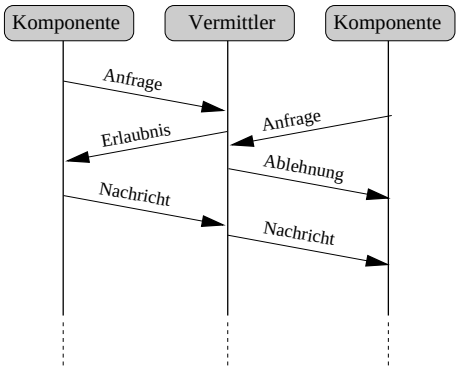


Abbildung 6.2 *Synchronisierter Kommunikationsablauf mit Vermittler-Komponente*

Ein Komponentenentwickler muss bei diesem Konzept dafür sorgen, dass ein solches Ereignis immer einem `SignaturEvent` vorausgeht und dass im Falle des Scheiterns einer Anfrage der interne Zustand der Komponente seiner Repräsentation im endlichen Automaten des zugehörigen `Extension`-Objektes entspricht. Gegebenenfalls müssen daher interne Zustandsveränderungen, die mit der gewünschten zu sendenden Nachricht in Zusammenhang stehen, rückgängig gemacht werden (engl. *rollback*). Im Rahmen dieser Arbeit wurde daher von [Lapok, 2000] eine transaktionale Absicherung der Komponentenkopplung realisiert.

6.1.3 Kompatibilitätsprüfung

In der hier implementierten Systemumgebung ist auch der in Abschnitt 3.4 präsentierte Kompatibilitätsprüfungsalgorithmus umgesetzt.[5] Dieser Algorithmus wird in der Klasse `Protocol_Tester` implementiert, die im Folgenden in Auszügen vorgestellt und erläutert wird.

[5] Der implementierte Algorithmus weicht in seinem Ablauf aus programmiertechnischen geringfügig von dem in Abschnitt 3.4 präsentierten Algorithmus ab.

6.1.3.1 Initialisierung

Über die jeweiligen `Extension`-Objekte der beiden Komponenten, deren Kompatibilität geprüft werden soll, ist ein direkter Zugriff auf den basierend auf den Angaben des Komponentenentwicklers (in einem Objekt vom Typ `MachineDescriber`) erzeugten endlichen Automaten möglich. Somit lässt sich zunächst das zu einem Automaten gehörende *Alphabet*, also die Menge aller Eingaben, die der Automat verarbeiten kann, aufbauen.

In diesem speziellen Fall baut `Protocol_Tester` ein gemeinsames Alphabet beider beteiligten Automaten auf – wobei lediglich Eingaben, die mit einem »–« beginnen (also zu sendende Nachrichten) von Bedeutung sind (s. Abschnitt 3.4; insbesondere Definition 3.4). Zudem werden Variablen initialisiert, aus denen sich am Ende des Algorithmus das Ergebnis der Prüfung synthetisieren lässt.

***Listing 6.6**: Die Klasse* `Protocol_Tester` *(Teil 1): Initialisierung*

```
public class Protocol_Tester {
  //globale Variablen
  StateMachine s1, s2;
  State initState1, initState2;
  Pair_of_States initPair;
  Vector equiv          = new Vector();
  Vector sendAlphabet    = new Vector();
  Vector receiveAlphabet = new Vector();
  boolean noNewElements            = false;
  //Bestandteile des R"uckgabewertes
  boolean hasUnspecifiedReceptions = false;
  boolean hasDeadlock              = false;
  boolean hasCollabs               = false;
  boolean reachedFinalPair         = false;
  boolean reachedInitPair          = false;

  //Konstruktor
  public Protocol_Tester (StateMachine s1, StateMachine s2) {
    this.s1 = s1;
    this.s2 = s2;
  }
  //Hauptmethode
  public int compatibility() {
    Pair_of_States p;
    Vector v = new Vector();
    boolean unset = false;
    int result = 0;
    //StateMachines (Automaten) auslesen
    //Anfangszustaende
    this.initState1 = s1.getInitState();
    this.initState2 = s2.getInitState();
    //send-alphabet entwickeln
    v = s1.getAlphabet();
    for (int i=0; i<v.size(); i++) {
      if ((((String)v.elementAt(i)).charAt(0)) == '-') {
```

```
        this.sendAlphabet.addElement(v.elementAt(i));
      }
    }
    v = s2.getAlphabet();
    for (int i=0; i<v.size(); i++) {
      if (((String)v.elementAt(i)).charAt(0) == '-') {
        unset = true;
          for (int j=0; j<this.sendAlphabet.size(); j++) {
            unset = unset && (! ((String)v.elementAt(i)).equals(
                              ((String)this.sendAlphabet.elementAt(j))));
          }
          if (unset) {
            this.sendAlphabet.addElement(v.elementAt(i));
          }
      }
    }
    //jetzt sollte jede send-message genau einmal
    //im zugehoerigen vector enthalten sein
...
```

6.1.3.2 Zustandspaare

Betrachtet werden seitens des Algorithmus Zustandspaare, die aus den jeweiligen Zuständen der beteiligten Komponenten zu bestimmten Zeitpunkten zusammengesetzt werden. Ein solches Zustandspaar wird durch Objekte vom Typ `Pair_of_States` beschrieben.

***Listing 6.7**: Die Klasse* `Pair_of_States`

```
public class Pair_of_States {
  State state1, state2;
  public Pair_of_States (State s1, State s2) {
    this.state1 = s1;
    this.state2 = s2;
  }
  public boolean equals (Pair_of_States p) {
    if ((p.state1.equals(this.state1)) && (p.state2.equals(this.state2)))
      { return true; }
    return false;
  }
}
```

6.1.4 Ablauf

Nach der Initialisierung wird das Anfangszustandspaar aus den ausgelesenen Anfangszuständen der beiden kollaborierenden Komponenten generiert und der neu erzeugten Menge `EQUIV` hinzugefügt. Mit diesem einzigen Element der Menge `EQUIV` wird erstmals die Suche nach Nachfolgezustandspaaren gestartet. Im weiteren Verlauf werden jeweils alle gefundenen neue Zustandspaare (d.h. Zustandspaare, die sich bisher nicht in `EQUIV` befanden) auf die Existenz von Nachfolgezustandspaaren hin untersucht. Dieser Ablauf wird beendet, wenn ausgehend von den

im vorigen (letzten) Durchgang neu gefundenen Zustandspaaren keine neuen Zustandspaare mehr gefunden werden.

***Listing 6.8**: Die Klasse* `Protocol_Tester` *(Teil 2): Beginn des Algorithmus*

```
...
//Anfangspaar in equiv-vector ablegen
this.initPair = new Pair_of_States(this.initState1, this.initState2);
this.equiv.addElement(this.initPair);
//nun geht es los...
v = check(equiv); //ersten Aufruf gesondert behandeln
//stattfindende Kommunikation vermerken!
if (! this.noNewElements) {this.hasCollabs=true;}
//alle weiteren mit Abfrage
while (! this.noNewElements) {
  v = check(v); //weitere Aufrufe mit neuen Elementen des letzten Aufrufes
}
...
```

Suche nach Nachfolgezustandspaaren Die Suche nach Nachfolgezustandspaaren erfolgt in der Methode `check` durch die sequentielle Prüfung des bei der Initialisierung erzeugten (Sende-) Alphabets auf die jeweiligen Automaten der beiden Komponenten unter Angabe des im Ausgangszustandspaar enthaltenen Zustandes. Die Bedingungen für das Erreichen eines gültigen Nachfolgezustandes finden sich in Abschnitt 3.6.2.

Das Auftreten eines unspezifizierten Empfangs lässt einen (einseitig) möglichen Übergang ungültig werden. Die Information über das Auftrteten eines unspezifizierten Empfangs wird gespeichert, damit sie später dem Resultat der Prüfung hinzugefügt werden kann. Auch die anderen Bestandteile des späteren Resultats werden geprüft und gegebenenfalls in ihrem Wert verändert.

***Listing 6.9**: Die Klasse* `Protocol_Tester` *(Teil 3): Die Methode* `check`

```
...
private Vector check(Vector e) {
  Pair_of_States p, pNext;
  Vector Next = new Vector();
  Vector elements = new Vector();
  boolean unset;
  boolean hasNew = false;
  boolean deadlock;
  Vector newElements = new Vector();
  //Vector kopieren
  for (int i=0; i<e.size(); i++) {
    elements.addElement(e.elementAt(i));
  }
  for (int i=0; i<elements.size(); i++) {
    //dieses Paar wird nun untersucht
    p = (Pair_of_States)elements.elementAt(i);
```

```
    deadlock = true; //zum Testen, ob dieses Paar in eine Sackgasse fuehrt
    for (int j=0; j<this.sendAlphabet.size(); j++) {
      Next = checkPair(p, (String)sendAlphabet.elementAt(j));
      if (Next != null) {
        //Nachfolgepaar(e) wurde(n) gefunden; es folgen einige Tests...
        deadlock = false;
        for (int g=0; g<Next.size(); g++) {
          pNext = (Pair_of_States)Next.elementAt(g);
          //schon in equiv?
          unset = true;
          for (int z=0; z<this.equiv.size(); z++) {
            unset = (unset && (!((Pair_of_States)this.equiv.
                                 elementAt(z)).equals(pNext)));
          }
          if (unset) { // nein
            this.equiv.addElement(pNext);
            newElements.addElement(pNext);
            hasNew = true; //damit es weitergehen kann...
          }
          //test auf final-states...
          if ((this.s1.isFinalState(pNext.state1))
              && (this.s2.isFinalState(pNext.state2))) {
            //erreichen eines endzustandspaares
            this.reachedFinalPair = true;
          }
          //test auf init-state
          if (pNext.equals(this.initPair)) {
             //Erreichen eines Anfangszustandspaares
             this.reachedInitPair = true;
          }
        } //for
      } //if
    } //for
    if (deadlock) {
    //wenn fuer p kein Nachfolger gefunden wurde
    //(und p kein finalPair ist), dann deadlock
      if (! ((this.s1.isFinalState(p.state1))
          && (this.s2.isFinalState(p.state2)))) {
        this.hasDeadlock = true; //deadlock wird vermerkt
      } //if
    } //if
  } //for
  //wenn keine neuen Elemente gefunden werden konnten
  if (! hasNew) { this.noNewElements = true; }
  return newElements;
}
...
```

6.1.4.1 Rückgabe des Resultats

Die beim Ablauf des Algorithmus gesammelten Informationen über die Existenz mindestens eines Kommunikationsschrittes, das Erreichen von Anfangs- oder Endzuständen, das Auftreten einer Verklemmung und das Auftreten von unspezifiziertem Empfang werden zum Abschluss der Methode `compatibility` als Ergebnis zurückgegeben. Das Ergebnis wird dabei in Form eines 5-Bit Wertes repräsentiert. Die Interpretation dieses Rückgabewertes richtet sich nach den in Abschnitt 3.6.1 beschriebenen Gesichtspunkten.

***Listing 6.10**: Die Klasse* `Protocol_Tester` *(Teil 4): Auswertung*

```
...
//Auswertung...
if (this.reachedFinalPair)         { result += 1; }
if (this.reachedInitPair)          { result += 2; }
if (this.hasCollabs)               { result += 4; }
if (this.hasDeadlock)              { result += 8; }
if (this.hasUnspecifiedReceptions) { result += 16; }
//5-bit wert zurueckgeben
return result;
...
```

6.1.4.2 Einbinden der Kompatibilitätsprüfung

Wie in Abschnitt 6.1.2.2 erwähnt, soll der Aufruf des beschriebenen Kompatibilitätsprüfungsalgorithmus durch eine übergeordnete Komponente der Entwicklungsumgebung erfolgen, beispielsweise wenn in der Kompositionsphase einer Anwendung Komponenten durch Kopplung zusammengefügt werden. Im Falle des Scheiterns einer Kompatibilitätsprüfung bietet sich das Einbinden von spezialisiert auf eine bestimmte Kopplung entwickelten Adapterkomponenten an, so wie es das Typmanagement vorsieht (vgl. Kapitel 4).

6.1.5 Adapterkomponenten

Adapterkomponenten beinhalten sowohl die formale Beschreibung ihrer Funktionalität als auch Methoden zu deren Implementierung. Diesen Methoden fällt somit die Aufgabe zu, den deklarativen Teil der formalen Beschreibung einzulesen und das Kommunikationsverhalten inklusive potenziellem Speichern und Löschen von Parametern entsprechend umzusetzen.

6.1.5.1 Einbindung in Kommunikationsablauf

Eine derartige Adapterkomponente wird als Interzeptor [Griffel et al., 1996] zwischen die zu den beiden Komponenten gehörenden `Extension`-Objekte gesetzt (entsprechend Abbildung 6.3). Die Gesamtkonstruktion repräsentiert dann die interne Struktur des Konnektors. Somit ist sichergestellt, dass sämtliche von den Komponenten gesendeten Nachrichten zum Adapter umgeleitet und dort verarbeitet werden. Die vom Adapter gesendeten Nachrichten werden je nach Adressat von dem zu einer Komponente gehörenden `Extension`-Objekt empfangen und dort als eingehende Nachricht verarbeitet.

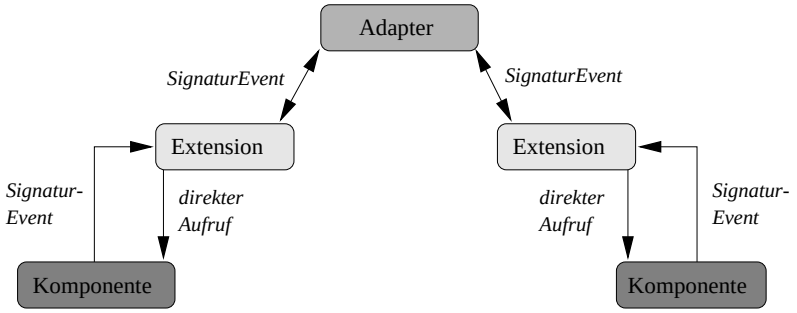


Abbildung 6.3 *Einbindung des Adapters in den Kommunikationsablauf*

Zur Einbindung der Adapterkomponente ist es erforderlich, dass die Adapterkomponente das Interface `SignaturListener` implementiert und bei den beteiligten `Extension`-Objekten entsprechend registriert wird.

6.1.5.2 Deklarativer Teil

Als Beispiel für einen derartigen Adapters soll nun eine Implementierung des in Abschnitt 3.7.5 beschriebenen Adapters betrachtet werden, der in der Lage ist, den zu den Kollaborationsspezifikationen der Beispiele 3.1 und 3.5 gehörenden Komponenten eine korrekte Kommunikation zu ermöglichen.

Listing 6.11*: Die Klasse* `Adapter` *(Teil 1): Regeln*

```
public class Adapter implements SignaturListener{
  //Adapterbeschreibung+++++++++++++++++++
  String init = "A,1";

  String[] trans =
    { "A,1", "B,1", "+newItem", "Auctioneer", "itemDescr,bidderId", "",
      "B,1", "B,2", "-auctionBegin", "Bidder", "itemDescr", "itemDescr",
      "B,2", "B,3", "+requestToBid", "Bidder", "bidItem", "",
      "B,2", "C,2", "+update", "Auctioneer", "highBid", "",
      "B,3", "B,4", "-canBid", "Bidder", "bidderId", "bidderId",
      "B,4", "C,7", "+update", "Auctioneer", "highBid", "",
      "B,4", "B,6", "+newBid", "Bidder", "bidId,myBid", "",
      "C,7", "C,5", "-newHighBid", "Bidder", "highBid", "highBid",
      "C,5", "C',4'", "+highBidAck", "Bidder", "", "", "",
      "C',4'", "B,4", "-updateAck", "Auctioneer", "", "",
      "B,6", "D,6", "-bid", "Auctioneer", "bidId,bidItem,myBid",
                                          "bidId,myBid",
      "D,6", "B',6'", "+rejectBid", "Auctioneer", "", "",
      "D,6", "E,6", "+acceptBid", "Auctioneer", "", "",
      "B',6'", "B,4", "-bidNotOk", "Bidder", "", "",
      "E,6", "E,7", "-bidOk", "Bidder", "", "",
      "E,7", "C,7", "+update", "Auctioneer", "highBid", "",
      "C,2", "B',2'", "-updateAck", "Auctioneer", "", "",
```

```
      "B',2'", "C,2", "+update", "Auctioneer", "highBid", "",
      "B',2'", "B',3'", "+requestToBid", "Bidder", "bidItem", "",
      "B',3'", "B',4'", "-canBid", "Bidder", "bidderId", "bidderId",
      "B',4'", "B,5", "-newHighBid", "Bidder", "highBid", "highBid",
      "B,5", "B,4", "+highBidAck", "Bidder", "", ""};
...
```

Ähnlich der in Abschnitt 6.1.1.1 beschriebenen Implementierung einer Kollaborationsspezifikation wird das Adapterprotokoll in einem Variablenfeld gespeichert. Nach Angabe des Anfangszustandes des auf Grund des Protokolls zu erzeugenden Automaten folgt im Feld `trans` eine Auflistung der möglichen Zustandsübergänge dieses Automaten in Sechsergruppen. Die dabei zu beachtende Reihenfolge lautet:

- Ausgangszustand,
- Zielzustand,
- Methodenname (inklusive Kommunikationsrichtung: »+« für zu empfangende und »–« für zu sendende Methoden),
- Name des Senders (bei Nachrichtenempfang) bzw. Empfängers (bei Nachrichtenversand) der der Nachricht (des Methodenaufrufs),
- Parameterbezeichner von zu speichernden (bei Empfang) bzw. auszulesenden (bei Versand) Parameterwerten,
- Parameterbezeichner von zu löschenden Parametern.

Durch Angabe dieser sechs Parameter ist ein Kommunikationsschritt eines Adapters vollständig beschrieben. Die einzelnen Zustände sind nach Beispiel 3.6 auf Seite 154 aus den Zuständen der beteiligten Komponenten zusammengesetzt, was in bestimmten Zuständen des Adapters nicht ausreicht – daher findet sich in Beispiel 3.6 auf Seite 154 und Listing 6.11 auf der vorherigen Seite ein Zustand mit der Bezeichnung »C',4'«, da dieser Zustand nicht mit dem Zustand mit der Bezeichnung »C,4« identisch ist. Theoretisch ist die konkrete Bezeichnung der Zustände beliebig und muss sich somit nicht an den Zustandsbezeichnern der beteiligten Komponenten orientieren.

6.1.5.3 Implementation der Funktionalität des Adapterprotokolls

Zu beachten ist, dass der Adapter sowohl die Namen als auch die Werte von bei einzelnen Methodenaufrufen übergebenen

Parametern benötigt. Der bei einem Kommunikationsvorgang übergebene `SignaturEvent` (s. Abschnitt 6.1.1.2) enthält im Feld für die Parameter zu diesem Zweck Objekte des Typs `Param_wrapper`, die wiederum die Bezeichner und das Parameter-Objekt enthalten.

***Listing 6.12**: Die Klasse* `Param_wrapper`

```
class Param_wrapper {
  String name;
  Object obj;
  Class type;

  public Param_wrapper(String name, Object obj) {
    this.name = name;
    this.obj  = obj;
    this.type = obj.getClass();
  }
    ...
```

Im Gegensatz zum Protokoll einer Kollaborationsspezifikation, das lediglich die interne Funktionalität einer Komponente beschreibt, dient ein Adapterprotokoll als Grundlage der Funktionalität der Adapterkomponente (vgl. Abschnitt 3.7.4). Somit muss die im Folgenden weiter erläuterte Adapterkomponente sich in ihrem Kommunikationsverhalten exakt am Adapterprotokoll orientieren. Das Adapterprotokoll dient als Vorlage für einen endlichen Automaten. Auf diesem Automaten werden zur Laufzeit sämtliche Eingaben (also empfangene oder zu sendende Nachrichten) geprüft und Zustandsänderungen durchgeführt.

Hierbei ist auf die in Abschnitt 3.7.6 beschriebene Problematik im Kommunikationsverhalten zu achten. Zur Ausführung der durch das Protokoll festgelegten Funktionalität führt ein Adapter, dessen Protokoll sich in einem Zustand befindet, der das Senden einer Nachricht ermöglicht, diesen Sendevorgang durch, ohne einen möglichen weiteren Nachrichtenempfang abzuwarten. Daher wird sowohl nach dem Empfang als auch nach dem Senden einer Nachricht geprüft, ob sich in dem dann erreichten Zustand des Adapterprotokolls eine Nachricht senden lässt. Ist dies der Fall, so wird diese Nachricht gesendet.

***Listing 6.13**: Die Klasse* `Adapter` *(Teil 2): Funktionalität*

```
...
MachineDescription target;
Class targetclass;
MachineDescriber md;
Simulator_extended s;
String[] transmessages;
Method[] targetmethods;
```

```
String lastGrantedRequest = "";
//hashtable zum Speichern der extension-objekte, die sich registrieren
private Hashtable listeners = new Hashtable();
//hashtable zur Speicherung von Parametern
private Hashtable params = new Hashtable();
//einlesen des protokolls
public Adapter () {
  int count = 0;
  transmessages = new String[trans.length];
  //einlesen der Methodenbezeichner
  while (count < trans.length) {
    count += 2;
    transmessages[count/3] = trans[count];
    count += 4;
  }
  //erzeugung des automaten
  s = new Simulator_extended();
  //Simulator_extended verarbeitet
  //Angaben aus Variablenfeld trans
  s.buildMachine(init, finals, trans);
}
//Nachrichtenempfang
public void transfer(SignaturEvent e) {
  Action_param ap; //fuer Antwort des Automaten
  Vector save_actions, load_actions, invalidate_actions;
  String methodname = e.method.substring(1);
  String from = e.from.getName(); //Absender auslesen
  //Format wird vom Automaten verlangt
  String method = new String(e.method+"::"+from);
  //Parameter werden verpackt
  Param_wrapper[] pw = new Param_wrapper[e.number_of_param];
  for (int i=0; i<e.number_of_param; i++) {
    pw[i] = (Param_wrapper)e.param[i];
  }
  //Automat gibt Methodenname, load- bzw. save-
  //und invalidate-actions zurueck
  ap = s.step(method);
  ...
  //hier jetzt save- und invalidate-actions ausfuehren
  save_actions = ap.loadsave_actions;
  invalidate_actions = ap.invalidate_actions;
  //erst invalidate...
  if (! ((String)invalidate_actions.firstElement()).equals("")) {
    for (int i=0; i<invalidate_actions.size(); i++) {
      String s = (String)invalidate_actions.elementAt(i);
      params.remove(s);
    }
  }
  //dann save
  if (! ((String)save_actions.firstElement()).equals("")) {
    for (int i=0; i<save_actions.size(); i++) {
      String s = (String)save_actions.elementAt(i);
      Param_wrapper p = null;
      for (int z=0; z<pw.length; z++) {
        if (pw[z].equals(s)) {
          p =pw[z];
          params.put(s,p);
        }
```

```
      }
    }
  }
  //nun muss geprueft werden, ob eine Methode gesendet werden soll
  //dazu dient Methode check
  String met = s.check();
  while (met != null) {
    send(met);
    met = s.check();
  }
}
```

Die hier nicht aufgelistete Methode `send` ist der Implementierung der Methode `transfer` fast identisch. Es ist dort keine Speicherung von Daten notwendig. Stattdessen werden gegebenenfalls gespeicherte Daten ausgelesen. Der damit konstruierte Methodenaufruf wird dann als Ereignis an den Adressaten gesendet. Auch nach Durchführung der Methode `send` wird als nächstes die Existenz weiterer zu sendender Nachrichten geprüft.

6.1.6 Kompatibilitätsprüfung für Adapter

Analog der in Abschnitt 6.1.3 präsentierten Implementierung einer Kompatibilitätsprüfung zweier Komponenten lässt sich die Kollaboration zweier Komponenten mit einer Adapterkomponente auf Kompatibilität hin überprüfen (s. Abschnitt 3.8).

6.1.6.1 Zustandstripel

Eine entsprechende Implementierung, wie sie sich hier in der Klasse `Adapter_Tester` findet, ist der bereits oben beschriebenen Implementierung der Klasse `Protocol_Tester` sehr ähnlich.

Basis der Kompatibilitätsprüfung und zugleich Elemente der während des Durchlaufs des Algorithmus erzeugten Mengen sind nun nicht mehr Objekte vom Typ `Pair_of_States` (Zustandspaare), sondern Objekte vom Typ `Triple_of_States` (Zustandstripel). Neben dem jeweiligen Zustand der beteiligten Komponenten wird nun auch der Zustand des Adapters einbezogen.

***Listing 6.14**: Die Klasse* `Triple_of_States`

```
public class Triple_of_States {
  State state1, state2, stateA;

  public Triple_of_States (State s1, State s2, State stateA) {
    this.state1 = s1;
    this.state2 = s2;
    this.stateA = stateA;
```

```
  }
  public boolean equals (Triple_of_States t) {
    if ((t.state1.equals(this.state1))
        && (t.state2.equals(this.state2))
        && (t.stateA.equals(this.stateA))) { return true; }
    return false;
  }
}
```

6.1.6.2 Ablauf

Die Implementierung des Algorithmus ist der Implementierung der Kompatibilitätsprüfung (s. Abschnitt 6.1.3) nahezu identisch. Basis der erzeugten Mengen sind nun Zustandstripel. Bei der Suche nach einem Übergang von einem Zustandstripel zu einem anderen wird zunächst die Adapterkomponente, potenziell gefolgt von einer der beiden beteiligten Komponenten (nämlich dem Adressaten der beim Adapter zu einem Zustandsübergang führenden Methode) und dann nacheinander beiden Komponenten potenziell gefolgt von der Adapterkomponente, unter sequentieller Abfrage aller möglichen zu sendenden Nachrichten (Methoden) auf einen Zustandsübergang geprüft.

Eine Kommunikation direkt zwischen beiden Komponenten unter Ausschluss der Adapterkomponente ist nicht vorgesehen und wird daher auch nicht überprüft.

***Listing 6.15**: Die Methode* `check` *aus der Klasse* `Adapter_Tester`

```
private Vector check(Vector e) {
  Triple_of_States t, tNext;
  Vector Next = new Vector();
  Vector elements = new Vector();
  boolean unset, deadlock;
  boolean hasNew = false;
  Vector newElements = new Vector();
  //Vector kopieren
  for (int i=0; i<e.size(); i++) {
    elements.addElement(e.elementAt(i));
  }
  for (int i=0; i<elements.size(); i++) {
  //dieses triple wird nun untersucht
  t = (Triple_of_States)elements.elementAt(i);
  //zum testen, ob dieses triple in eine Sackgasse f"uhrt
  deadlock = true;
  for (int j=0; j<this.sendAlphabet.size(); j++) {
    Next = checkTriple(t, (String)sendAlphabet.elementAt(j));
      if (Next != null) {
        //nachfolgepaar(e) wurde(n) gefunden; es folgen einige tests...
        deadlock = false;
        for (int g=0; g<Next.size(); g++) {
          tNext = (Triple_of_States)Next.elementAt(g);
          //schon in equiv?
```

```
          unset = true;
          for (int z=0; z<this.equiv.size(); z++) {
          unset = (unset && (! ((Triple_of_States)this.equiv.
                              elementAt(z)).equals(tNext)));
          }
          if (unset) { // nein
            this.equiv.addElement(tNext);
            newElements.addElement(tNext);
            hasNew = true; //damits weitergehen kann...
          }
          //test auf final-states...
          if ((this.s1.isFinalState(tNext.state1))
              && (this.s2.isFinalState(tNext.state2))
              && (this.sa.isFinalState(tNext.stateA)) ) {
            this.reachedFinalTriple = true;
          }
          //test auf init-state
          if (tNext.equals(this.initTriple)) {
            this.reachedInitTriple = true;
          }
        } //for
      } //if
    } //for
    if (deadlock) {
      if (! ((this.s1.isFinalState(t.state1))
          && (this.s2.isFinalState(t.state2))
          && (this.sa.isFinalState(t.stateA)))) {
        this.hasDeadlock = true;
      } //if
    } //if
  } //for
  //abbruch, wenn keine neuen elemente gefunden werden konnten
  if (! hasNew) { this.noNewElements = true; }
  return newElements;
}
```

6.1.6.3 Rückgabe des Resultats

Der von der (hier nicht aufgelisteten) Methode `compatibility` zurückgegebene 5-Bit-Wert entspricht in seiner Zusammensetzung dem von der Kompatibilitätsprüfung (s. Abschnitt 6.1.4.1) zurückgegebenen Wert. Dabei ist ein Auftreten eines unspezifizierten Empfangs nur von Bedeutung, wenn er bei einer der beiden beteiligten Komponenten vorkommt. Unspezifizierter Empfang seitens der Adapterkomponente ist auf Grund der hier vorgenommenen Prüfung nach dem Kriterium der *schwachen* Kompatibilität (vgl. Abschnitt 3.8.2) erlaubt.

6.1.7 Semiautomatische Generierung von Adapterprotokollen

Dass in Abschnitt 3.9 präsentierte Konzept der semiautomatischen Adaptergenerierung wird durch die Klasse `Generate_Adapter` implementiert. Die zur Erzeugung eines Adapterprotokolls notwendigen Metainformationen werden in der

Klasse `Meta_Adapter` spezifiziert oder bei vorhandensein der Typmanagment-Komponente von dieser erfragt. Mit dem generierten Adapterprotokoll wird anschließend ein Objekt vom Typ `Adapter` parametrisiert.

6.1.7.1 Die Klasse `Meta_Adapter`

Für die halbautomatische Generierung eines Adapterprotokolls notwendige Metainformationen bestehen aus `param_mapping_rules`, `one_shot_rules` und `causality_rules`. Als Container für diese Informationen dient die Klasse `Meta_Adapter`, die lediglich aus multidimensionalen Variablenfeldern besteht, die die von `Generate_Adapter` zu verarbeitenden Informationen bereitstellen. Als Beispiel für eine mögliche Implementierung von `Meta_Adapter` soll eine Bereitstellung der das Adapterprotokoll aus Beispiel 3.7 beschreibenden Regeln betrachtet werden.

***Listing 6.16**: Die Klasse* `Meta_Adapter`

```
public class Meta_Adapter {
  //Adapterbeschreibung+++++++++++++++++++
  //one-shot-rules
  String[][] oneshot_param_mapping =
    {{"Bidder::newBid.bidId", "Auctioneer::bid.bidderId"},
     {"Bidder::newBid.myBid", "Auctioneer::bid.amount"},
     {"Auctioneer::newItem.itemDescr", "Bidder::auctionBegin.auctionInfo"},
     {"Auctioneer::newItem.bidderId", "Bidder::canBid.bidId"},
     {"Auctioneer::update.highBid", "Bidder::newHighBid.price"}};
  //multi-shot-rules
  String[][] multishot_param_mapping =
    {{"Bidder::requestToBid.bidItem", "Auctioneer::bid.itemBiddingOn"}};

  int nocr = 9; //Anzahl der causality-rules
  MachineDescriber[] md = new MachineDescriber[nocr];
  //sende-alphabet
  String[] sendmsg = {"-auctionBegin", "-newHighBid", "-bidNotOk",
                      "-bidOk", "-gavel", "-updateAck", "-canBid", "-bid"};
  //anfangszustaende der causality-rules protokolle
  String[] init = {"A", "A", "A", "A", "A", "A", "A", "A", "A"};
  //endzustaende (ohne bedeutung)
  String[][] finals = { {"B"}, {"B"}, {"B"}, {"B"}, {"D"},
                        {"B"}, {"D"}, {"B"}, {"B"} };
  //causality-rules als protokolle
  String[][] trans   = {{ "A", "B", "+newItem::Auctioneer",
                          "B", "A", "-auctionBegin::Bidder"},

                        { "A", "B", "+rejectBid::Auctioneer",
                          "B", "A", "-bidNotOk::Bidder"},

                        { "A", "B", "+acceptBid::Auctioneer",
                          "B", "A", "-bidOk::Bidder"},
```

```
                        { "A", "B", "+itemSold::Auctioneer",
                          "B", "A", "-gavel::Bidder"},

                        { "A", "B", "+update::Auctioneer",
                          "B", "C", "+reqToBid::Bidder",
                          "C", "D", "-newHighBid::Bidder",
                          "D", "A", "+itemSold::Auctioneer"},

                        { "A", "B", "+highBidAck::Bidder",
                          "B", "A", "-updateAck::Auctioneer"},

                        { "A", "C", "+reqToBid::Bidder",
                          "A", "B", "+update::Auctioneer",
                          "B", "A", "-updateAck::Auctioneer",
                          "C", "D", "+update::Auctioneer",
                          "C", "A", "+itemSold::Auctioneer",
                          "D", "C", "-newHighBid::Bidder"},

                        { "A", "B", "+reqToBid::Bidder",
                          "B", "A", "-canBid::Bidder"},

                        { "A", "B", "+newBid::Bidder",
                          "B", "A", "-bid::Auctioneer"}};

  public Meta_Adapter() {
    //hier MachineDescriber erzeugen...
    for (int i=0; i<nocr; i++) {
      md[i] = new MachineDescriber(init[i], finals[i],
                                   trans[i], null, null);
    }
  }
}
```

Die Felder `oneshot_param_mapping` und `multishot_param_mapping` bestehen jeweils aus Subfeldern mit zwei Elementen, von denen das mit dem Index 0 (links) den Ausgangsparameter und das Element mit Index 1 (rechts) den mit dem Wert des Ausgangsparameters zu erzeugenden Parameter festlegt. Die `causality_rules` werden als endliche Automaten implementiert. Zu diesem Zweck wird für jede `causality_rule` ein eigenes Protokoll (syntaktisch identisch zur Protokollspezifikation einer Komponente) festgelegt, das dann in Objekten vom Typ `MachineDescriber` gespeichert wird.

Die in Objekten vom Typ `Meta_Adapter` derart bereitgestellten Informationen können nun von der Klasse `Generate_Adapter` zur Generierung eines Adapterprotokolls verwendet werden.

6.1.7.2 Die Klasse `Generate_Adapter`

`Generate_Adapter` generiert ein Adapterprotokoll, falls dies möglich ist – ansonsten wird ein Adapterprotokoll mit 0 Zeilen zurückgegeben. Der Ablauf des dazu implementierten Algorithmus folgt den in Abschnitt 3.9.3 beschriebenen Regeln und ähnelt damit in der grundsätzlichen Konzeption den Abläufen in den

Klassen `Protocol_Tester` und `Adapter_Tester`. Basis dieses Algorithmus sind Objekte vom Typ `Tuple_of_States`, die ein Zustandstupel repräsentieren.

6.1.7.3 Tuple_of_States

***Listing 6.17**: Die Klasse* `Tuple_of_States`

```
public class Tuple_of_States {
  State state1, state2;
  Vector memCells;
  State[] patternState;

  public Tuple_of_States (State s1, State s2, Vector memCells,
                          State[] patternState) {
    this.state1 = s1;
    this.state2 = s2;
    this.memCells = memCells;
    this.patternState = patternState;
  }
  ...
  //vergleich zweier zustandstupel
  //hierbei ist vergleich der einzelnen memcells noetig
  public boolean equals (Tuple_of_States p) {
    boolean tmp;
    if (! ((p.state1.equals(this.state1))
           && (p.state2.equals(this.state2)))) { return false; }
    if (! (p.memCells.size() == this.memCells.size())) { return false; }
    //Vector kopieren
    Vector vtmp = new Vector();
    for (int q=0; q < this.memCells.size(); q++) {
      vtmp.addElement(this.memCells.elementAt(q));
    }
    for (int z=0; z < p.memCells.size(); z++) {
      tmp = false;
      for (int q=0; q < vtmp.size(); q++) {
        if ((p.memCells.elementAt(z).equals(vtmp.elementAt(q))))
          { tmp = true; vtmp.removeElementAt(q); break; }
      }
      if (tmp == false)
        return false;
      }
      return true;
    }
    ...
}
```

Initialisierung Zur Initialisierung dient der Konstruktor `Generate_Adapter()`, dem die beteiligten `Extension`-Objekte sowie das Objekt vom Typ `Meta_Adapter` übergeben werden. Die beteiligten Komponentenprotokolle werden ausgelesen, ein entsprechender Automat erzeugt und initialisiert und ein Alphabet aus den potenziell zu sendenden Nachrichten gebildet. Ein solches Alphabet wird auch aus dem Objekt vom Typ `Meta_Adapter` ausgelesen. Zudem werden anhand der dort vorge-

gebenen Protokolle (entsprechend den `causality_rules`) Automaten erzeugt und initialisiert. Die Regeln zur Parameterverwendung werden gespeichert.

***Listing 6.18**: Die Klasse* `Generate_Adapter` *(Teil 1): Initialisierung und Beginn des Algorithmus*

```
public class Generate_Adapter {
  Vector LinesOfAdapter = new Vector();
  Vector tmpLines = new Vector();
  ...
  public Generate_Adapter(Extension auc, Extension bid) {
    int count;
    boolean unset = false;
    String tmp;
    Vector v = new Vector();
    Vector vtmp;
    Tuple_of_States tmp1, tmp2;
    //erstmal MachineDescriber einlesen und Simulator-Objekte erzeugen
    Meta_Adapter ms = new Meta_Adapter();
    this.oneshot_param_mapping = ms.oneshot_param_mapping;
    this.multishot_param_mapping = ms.multishot_param_mapping;
    this.nocr = ms.nocr;
    md = new MachineDescriber[nocr];
    for (int i=0; i<nocr; i++) {
      this.md[i] = ms.md[i];
    }
    ...
    //jetzt bidder und auctioneer einlesen...
    cmd[0] = auc.getMachineDescription().getMachineDescriber();
    cmd[1] = bid.getMachineDescription().getMachineDescriber();
    s1 = auc.getStateMachine();
    s2 = bid.getStateMachine();
    //Anfangszustaende
    this.initState1 = s1.getInitState();
    this.initState2 = s2.getInitState();
    //send-alphabet entwickeln
    ...
    //send-alphabet von meta-adapter
    String[] mas = ms.sendmsg;
    for (int i=0; i<mas.length; i++) {
      tmp = mas[i];
      this.sendAlphabetAdap.addElement(tmp);
    }
    //Initialisierung ist hiermit abgeschlossen
    //Es folgt: Erzeugung des Adapters A0
    Generate_A0();
    ...
```

Protokollgenerierung des Adapters `A0` Es wird entsprechend der Konzeption aus Abschnitt 3.9.3.2 zunächst ein Adapterprotokoll eines Adapters `A0` generiert, das schließlich sukzessiv zu einem korrekten Adapterprotokoll umgewandelt wird.

***Listing 6.19**: Die Klasse* `Generate_Adapter` *(Teil 2): Die Methode* `Generate_A0`

```
public void Generate_A0() {

  Tuple_of_States t;
```

```
  Vector v = new Vector();
  boolean unset = false;
  int result = 0;
  String tmp, m;
  Line_of_Adapter vtmp;
  //anfangstupel
  this.initTuple = new Tuple_of_States(this.initState1, this.initState2,
                                       new Vector(), cStates);
  this.equiv.addElement(this.initTuple);
  //nun geht es los...
  //ersten aufruf gesondert behandeln...
  v = check(equiv);
  for (int i=0; i<equiv.size(); i++) {
    t = (Tuple_of_States)equiv.elementAt(i);
  }
  while (! this.noNewElements) {
    for (int i=0; i<tmpLines.size(); i++) {
      vtmp = (Line_of_Adapter)this.tmpLines.elementAt(i);
      //schon in LinesOfAdapter?
      //(LinesOfAdapter ist ein Vektor der bisher
      //gefundene Zustandstupeluebergaenge enthaelt)
      unset = true;
      for (int z=0; z<this.LinesOfAdapter.size(); z++) {
        unset = (unset && (! ((Line_of_Adapter)this.
                            LinesOfAdapter.elementAt(z)).equals(vtmp)));
      }
      if (unset) { // nein
        LinesOfAdapter.addElement(vtmp);
        //tuple in sourcetuples speichern
        sourceTuples.addElement(vtmp.getSourceTuple());
      }
    }
    //jeder weitere aufruf mit neuen Elementen des letzten Aufrufes
    v = check(v);
    for (int i=0; i<equiv.size(); i++) {
      t = (Tuple_of_States)equiv.elementAt(i);
    }
    for (int i=0; i<v.size(); i++) {
      t = (Tuple_of_States)v.elementAt(i);
    }
  }
  //den letzten durchgang speichern
  for (int i=0; i<tmpLines.size(); i++) {
    vtmp = (Line_of_Adapter)this.tmpLines.elementAt(i);
    //schon in LinesOfAdapter?
    unset = true;
    for (int z=0; z<this.LinesOfAdapter.size(); z++) {
      unset = (unset && (! ((Line_of_Adapter)this.LinesOfAdapter.
                            elementAt(z)).equals(vtmp)));
    }
    if (unset) { // nein
      LinesOfAdapter.addElement(vtmp);
      //tuple in sourcetuples speichern
      sourceTuples.addElement(vtmp.getSourceTuple());
    }
  }
  //in LinesOfAdapter befindet sich jetzt der komplette Adapter A0
}
```

Während des laufenden Algorithmus zur Generierung des Adapters A0, der in seiner Implementierung den Implementierungen der Algorithmen der Kompatibilitätsprüfung ähnelt, werden Mengen aus Zustandstupeln generiert. In der Menge EQUIV werden dabei sämtliche Zustandstupel gesammelt, die sich unter Berücksichtigung der Komponentenprotokolle sowie den Regeln der Meta-Adapterspezifikation nach den in Abschnitt 3.9.3.1 beschriebenen Bedingungen erreichen lassen.

Die Zustandstupel, die auf diese Weise gesammelt werden, reichen für die Erstellung eines Adapterprotokolls jedoch nicht aus. Zusätzlich zur Angabe von Ausgangs- und Zielzustand des Adapters (diese Informationen befinden sich in Tuple_of_States) wird die Angabe des verwendeten Methodenaufrufs sowie Informationen über zu ladende, zu speichernde oder zu löschende Parameterwerte benötigt.

Line_of_Adapter Die Speicherung dieser Informationen erfolgt beim Auffinden gültiger Zustandstupelübergänge in Objekten des Typs Line_of_Adapter.

***Listing 6.20**: Die Klasse* Line_of_Adapter

```
public class Line_of_Adapter {
  Tuple_of_States t1, t2;
  String message, loadsave, invalidate;

  public Line_of_Adapter (Tuple_of_States t1, Tuple_of_States t2,
                          String message, String loadsave,
                          String invalidate) {
    this.t1 = t1;
    this.t2 = t2;
    this.message = message;
    this.loadsave = loadsave;
    this.invalidate = invalidate;
  }
  ...
  public void setTargetState1 (State s) {
    t2.setState1(s);
  }
  public void setTargetState2 (State s) {
    t2.setState2(s);
  }
  public String toString() {
    String result = new String(t1.toString()+" "+t2.toString()
                               +"\nMessage:  "+message+"\nloadsave:  "
                               +loadsave+"\ninvalidate: "+invalidate);
    return result;
  }
...
```

```
}
```

Die Methode `toString()` wird am Ende des Algorithmus der Ausgabe der einzelnen Zeilen des Protokolls dienen.

Generierung des korrekten Adapterprotokolls Aus dem zu Adapter `A0` generierten Protokoll werden nun sukzessiv alle Zustandsübergänge entfernt, die in eine Verklemmung, d.h. in einem Zustandstupel enden, von dem aus es keinen gültigen Zustandstupelübergang gibt. Zu diesem Zweck werden die gesammelten `Line_of_Adapter`-Objekte (also das Protokoll insgesamt) dahingehend untersucht, ob der jeweils angegebene Zielzustand in irgendeiner Zeile des Protokolls auch als Anfangszustand angegeben ist. Ist dies nicht der Fall und handelt es sich bei dem betrachteten Zustandstupel auch nicht um ein aus den Endzuständen der beteiligten Komponenten zusammengesetztes Tupel, dann liegt eine Verklemmung vor und der Zustandsübergang wird entfernt.

Nachdem alle Zustandsübergänge des Protokolls in dieser Art untersucht und gegebenenfalls entfernt wurden, beginnt diese Prüfung mit dem nun entstandenen Protokoll noch einmal von vorn. Dieses wiederholt sich solange bis bei einem Durchgang kein Zustandsübergang entfernt wurde – also keine Verklemmung existiert. Die nun übrig gebliebenen Zustandsübergänge definieren das Protokoll eines zu den beteiligten Komponenten kompatiblen Adapters. Falls keine Zustandsübergänge übrig geblieben sind, konnte kein korrekter Adapter generiert werden. Die Implementierung dieses Algorithmus zur Erzeugung und der Ausgabe des korrekten Adapters zeigt Listing 6.21.

***Listing 6.21**: Die Klasse* `Generate_Adapter` *(Teil 3): Aufruf* `Generate_Ax` *und Resultatausgabe*

```
...
//In Methode Generate_Adapter
while (Generate_Ax()) {
//wiederholen bis kein remove noetig
}
//Ausgabe
for (int g=0; g<LinesOfAdapter.size(); g++) {
  System.out.println(
    ((Line_of_Adapter)LinesOfAdapter.elementAt(g)).toString());
}
...
public boolean Generate_Ax() {
  //entfernt deadlock-lines
  Tuple_of_States source, target;
  Line_of_Adapter vtmp;
  Vector removelist = new Vector();
```

```
  boolean deadlock;
  for (int z=0; z<this.LinesOfAdapter.size(); z++) {
    vtmp = (Line_of_Adapter)this.LinesOfAdapter.elementAt(z);
    target = (Tuple_of_States)vtmp.getTargetTuple();
    deadlock = true;
    //target auch in source?
    for (int i=0; i<this.sourceTuples.size(); i++) {
      source = (Tuple_of_States)this.sourceTuples.elementAt(i);
        if (source.equals(target)) {
          deadlock = false; break; }
    }
    if (deadlock) {
      //test auf final-states...
      if ((this.s1.isFinalState(target.state1))
          && (this.s2.isFinalState(target.state2))) {
        //nicht entfernen!
      }
      else {
        removelist.addElement(vtmp);
      }}}
  if (removelist.isEmpty()) {
    return false;
  }
  for (int z=0; z<removelist.size(); z++) {
    this.LinesOfAdapter.removeElement(removelist.elementAt(z));
    vtmp = (Line_of_Adapter)removelist.elementAt(z);
    this.sourceTuples.removeElement(vtmp.getSourceTuple());
  }
  return true;
}
```

Die auf diese Weise implementierten Algorithmen bzw. auch die Adapterkomponenten selbst fließen nun in die Realisierung des Typmanagements zur Überprüfung bzw. Bereitstellung einer Kompatibilität ein. Die entsprechende Implementation eines *Typmanagers* und seiner interaktiven Bedienung wird im Weiteren dargestellt.

6.2 Der Typmanager

Das in Kapitel 4 vorgestellte Klassifikationskonzept resultiert in Systemanforderungen an eine Typmanagement-Komponente. Diese werden im Folgenden zunächst konzeptionell, und somit noch von einer konkreten Technologie losgelöst vorgestellt und anschließend mittels konkreter Java-Technologien prototypisch umgesetzt.

Ein *Typmanager* ist eine Applikation, die die Registrierung und Verwaltung von Typbeschreibungen in Form des formalisierten Typmodells erlaubt. Zum Registrierungszeitpunkt wird die gegebene Typbeschreibung im ersten Schritt auf die Typmanager-interne Repräsentation des Komponententyps (Typbaum) abgebildet, sodass im Anschluss daran deren Einordnung in die Gesamthierarchie aller Komponententypen vor-

genommen werden kann. Da hinsichtlich der Einordnung des Komponententyps sämtliche Beziehungen zu allen in der jeweiligen Domäne registrierten Komponententypen evaluiert werden, kann dieser Schritt möglicherweise sehr zeitaufwändig sein. Allerdings ist er für jede Komponente einmalig und ermöglicht nachfolgend effiziente Operationen zur Typsuche.

Im Hinblick auf die Evaluierung von Beziehungen zwischen Komponententypen stellt der Typmanager Operationen für den Typvergleich zur Verfügung und erlaubt Anfragen, um potenzielle Interaktionspartner auf Kompatibilität zu prüfen. Der Typvergleich basiert dabei auf einer Evaluierung von Gleichheits-, Basistyp- sowie Typart-Beziehungen, auf deren Grundlage eine Aussage über die Konformität des spezifizierten Substitutionskandidaten abgeleitet werden kann.

Durch diese Basisfunktionalität kann der Typmanager eine Wissensbasis aufbauen, die zur Laufzeit stets erweitert werden kann. Die Wissensbasis stützt sich dabei auf die bereits evaluierten Beziehungen zwischen den Komponententypen der Domäne in Form von Assoziationen zwischen deren Typbäumen. Es entsteht somit eine Art *Assoziationsgeflecht* zwischen Typrepräsentationen in enger Analogie zum aktuellen ISO-Standard *Topic Maps*[6], wobei eine Navigation durch die aufgebaute Netzstruktur möglich ist [Rath, 1999]. Eine Erweiterung der Wissensbasis findet dann stets bei der Registrierung eines Komponententyps statt. Somit lassen sich darauf aufbauend ebenso Anfragen stellen, die bestimmte Typausprägungen (Basistypen, Typarten) sowie potenzielle Interaktionspartner (kompatible Typen) zu einem selektierten Komponententyp zurückgeben. Ferner kann eine Funktion zur Adaptersuche bereitgestellt werden, die bei Bedarf einen geeigneten Adaptionsmechanismus für zwei zu koppelnde, inkompatible Komponenten automatisch zuordnet.

Letztendlich erlaubt die Typmanagement-Komponente die Verwaltung hierarchisch strukturierter Typdomänen, um verschiedene Namensräume voneinander abzugrenzen. Assoziationen werden dabei lediglich innerhalb der einzelnen Typdomänen berechnet, sodass für jeden Namensraum eine individuelle Wissensbasis in Form einer Topic Map etabliert wird.

Die folgenden Abschnitte dokumentieren die Implementation des in Kapitel 4 vorgeschlagenen Typmodells sowie die Realisierung der zugehörigen Typmanagement-Komponente (Typmanager). Dabei liegt der Schwerpunkt auf der Darstellung der von

[6] Der ISO-Standard Topic Maps wurde im Herbst 1999 verabschiedet (ISO/IEC 13250:1999).

der Typmanagement-Komponente bereitgestellten Funktionalität. Ferner wird der Einsatz der interaktiven Nutzung in Form des *Type Manager Explorers* erläutert, der neben dem API (Application Programming Interface) des Typmanagers als grafische Benutzeroberfläche (GUI) zur Verfügung steht. Aus der Identifikation der verschiedenen Funktionsbereiche des Typmanagers folgt eine entsprechende Modularisierung der Implementation.

6.2.1 Typregistrierung

Vor der Ausführung von Operationen auf Typrepräsentationen muss im ersten Schritt zunächst die Registrierung entsprechender Komponententypen – also der Komponententypbeschreibungen – vorgenommen werden. Die aus der Modellierung hervorgegangenen Metainformationen der Komponente, für die nachfolgend XML als Repräsentationsformat genutzt wird, werden dabei vom Typmanager mit Hilfe eines XML-Parsers [Sun, 1999b] eingelesen und auf die in Kapitel 4 formalisierte interne Struktur des Komponententyps (Typbaum) abgebildet. Auf Basis dieser Typrepräsentationen können anschließend Typoperationen sowie Funktionen zur Typsuche durchgeführt werden. Komponenten müssen nicht im Sinne aktiver Instanzen vorliegen: Um ihren entsprechenden Typ zu registrieren kann sowohl eine manuelle als auch eine durch die Komponenten selbst initiierte Anmeldung beim Typmanager erfolgen (vgl. a. das Konzept eines Traders [Müller et al., 1994][AT&T et al., 1996][Griffel, 1998]).

6.2.1.1 Typbeschreibungen in XML

Für die Typbeschreibung einer Komponente wird die Extensible Markup Language (XML) [W3C, 1998] [Tolksdorf, 1999] heran gezogen. XML fordert dabei die Definition sogenannter *Document Type Definitions* (DTDs), die die entsprechenden Regeln (Grammatik) zum Aufbau von zu diesem *Dokumenttyp* konformen XML-Dokumenten festlegen. Ein XML-Parser kann daraufhin die Wohlgeformtheit einer XML-Spezifikation prüfen, indem er diese gegen das zu Grunde liegende Meta-Modell in Form der DTD abgleicht (validierendes Parsen).

Die Definition des vollständigen Komponenten-Meta-Modells wird dann nicht durch einen einzigen solchen Dokumenttyp, sondern vielmehr durch insgesamt fünf DTDs realisiert, um zum einen eine gewisse Übersichtlichkeit zu gewährleisten, zum anderen schon bereits durch die Dokumenttypen die Abgrenzung der einzelnen Ebenen des Typbaums vorzunehmen. Auf die Doc-

ument Type Definitions selbst soll im Folgenden nicht eingegangen werden (s.a. Anhang A). Stattdessen werden entsprechende XML-Deskriptoren exemplarisch vorgestellt.

Somit ist an dieser Stelle die Notwendigkeit einer Zusammenfassung aller zu *einer* Komponententypbeschreibung korrespondierenden XML-Deskriptoren gegeben, um von *der* Typbeschreibung sprechen zu können bzw. ihre einfache Handhabung zu erlauben. Diese Forderung kann durch ein *Komponentenarchiv* (Component Archive) realisiert werden. Für den exemplarisch definierten Komponententyp *Bank* sollen nachfolgend XML-Deskriptoren angegeben werden, die dem in Anhang D dargestellten Ausschnitt des Typbaums entsprechen. Für die semantische Ebene ist dies der `CorbaComponentDescriptor`, für die Nachrichtenebene der `SyntBETypeDescriptor` sowie der `SyntMessageTypeDescriptor` für die syntaktische Ebene, wobei der jeweilige XML-Deskriptor stets auf Basis einer DTD festgelegt wird. Für Protokolle und Merkmalsstrukturen werden dabei zusätzliche XML-Deskriptoren basierend auf zwei weiteren DTDs definiert (vgl. Anhang A). Dieser einheitliche Einsatz von XML zur Repräsentation aller Informationen ergibt eine homogene Verarbeitung und eine flexible Erweiterbarkeit des Modells. Darüber hinaus ist eher schlichtweg pragmatisch motiviert, da entsprechende Deskriptoren bereits für Werkzeuge und Architekturen der industriellen Entwicklungspraxis definiert sind und so eine Abwärtskompatibilität zu bestehenden Werkzeugen, Entwicklungsumgebungen bzw. Komponentenmodellen erreicht werden kann. Im Folgenden soll daher kurz auf diese existierenden Deskriptoren-Konzepte eingegangen werden.

6.2.1.2 XML-Deskriptoren

Bis vor etwa drei Jahren wurde bei Spezifikationen von Komponenten besonderes Augenmerk darauf gelegt, zu beschreiben, wie auf die Funktionalität der Komponente im Sinne einer API zugegriffen werden kann. Aber neben Informationen über diese syntaktischen Aspekte forderte auch die Praxisicht zunehmend die definierte Bereitstellung von zusätzlichen Informationen die die Semantik der Komponente oder ihre Einsatzsituation betreffen. Auch Abhängigkeiten zu anderen Komponenten sollten ausgedrückt werden können, da diese ja gemäß der Komponentensicht eben nicht *in* einer Komponente codiert werden, sondern als explizite externe Information den jeweiligen Baustein begleiten sollen. Es gibt inzwischen einige Bestrebungen Komponentenspezifikationen in diesem Sinne zu standarisieren. Die Beschreibung

sollte dabei unabhängig von der Implementierungsart / -technik der Komponente sein. Ausserdem sollte sie plattformunabhängig erfolgen. Die eXtensible Markup Language (XML)[7] als derzeit beliebtestes und sowohl auf Grund ihrer hohen Akzeptanz als auch auf Grund ihrer wenigen Vorgaben universelles Beschreibungsmittel ist auch in diesem Kontext in den Mittelpunkt entsprechender praxisgetriebener Bemühungen geraten.

Die XML ist letztlich eine sauber definierte Repräsentation einer kontextfreien Grammatik in deren Rahmen ein beliebiges Vokabular definiert werden kann (sogenannte *tags*), die es erlauben ein Textdokument mit zusätzlicher Semantik anzureichern im Sinne einer Auszeichnungssprache (engl. *markup language*). Das bedeutet, dass XML-Dokumente einfache plattformunabhängige Textdateien sind, die sowohl von Menschen als auch von einem Computer gelesen und verstanden werden können. XML zeigt seine Stärken in der Speicherung von strukturierten Daten, die zwischen Anwendungen ausgetauscht werden können. Die Informationen aus dem XML-Dokument müssen durch einen Parser interpretiert werden. Es gibt keinen festen Satz an Tags wie etwa bei HTML. Wie das Wort »extensible« andeutet, handelt es sich vielmehr um eine Sprache, die erweitert kann. Durch diese Erweiterungsmechanismen ist XML sehr ausdrucksmächtig. Es wird nur festgelegt, wie die Grundstruktur eines XML-Dokumentes aussehen soll. Mit Hilfe von Document Type Definitions (DTD) ist es möglich, Klassen von Dokumenten zu definieren.

Open Software Description Die Firmen Marimba und Microsoft haben 1997 dem World Wide Web Consortium (W3C) gemeinsam einen Vorschlag zur Standarisierung der Beschreibung von Softwarekomponenten unterbreitet [van Hoff et al., 1997]. Die Beschreibung ist dabei unabhängig von der verwendeten Komponententechnologie. Der Vorschlag beruht auf der Verwendung von XML als plattformunabhängiger und erweiterbarer Beschreibungsmöglichkeit. Das Beschreibungskonzept wird entsprechend als Open Software Description (OSD) bezeichnet. Das Hauptaugenmerk liegt auf der Beschreibung von Software*paketen* zwecks deren Auslieferung (speziell via Internet). Es bietet die Möglichkeit eine Art Inhaltsverzeichnis von Komponentenarchiven anzulegen. Es ist ausserdem möglich, andere Softwareeinheiten zu beschreiben, die auf dem jeweilgen Rechner für die Verwendung einer Komponente benötigt werden.

[7] `http://www.wC3.org/XML`

OSD soll den Download, die Überprüfung von Zertifizierungen und die Installation von Softwarekomponenten erleichtern. Der Microsoft Internet Explorer etwa unterstützt seit Version 4 OSD. Eine Software kann aufgeteilt werden in unterschiedliche Pakete für einzelne Betriebssysteme und Landessprachen. Der Download einer Komponente wird optimiert, indem nur die wirklich benötigte Version heruntergeladen wird. Dies ist duch die Benutzung mehrerer IMPLEMENTATION-Tags möglich, die wiederum durch Angabe der Tags OS (Operating System, d.h. Betriebssystem), PROZESSOR und LANGUAGE (Landessprache) auf unterschiedliche Pakete verweisen können.

OSD beschreibt die Abhängigkeiten zwischen Teilen einer Software als gerichtete Graphen. Die modulare Bauweise von Software bewirkt, dass eine Komponente auf das Vorhandensein anderer Komponenten angewiesen sein kann (vgl. Kapitel 2). In einem solchen Fall kann überprüft werden, ob diese Komponenten zum Beispiel beim herunterladenden Client vorhanden sind und gegebenenfalls werden fehlende Komponenten ebenfalls heruntergeladen. Dies kann automatisch erfolgen, da durch die Angabe von URLs direkt auf diese Komponenten im World Wide Web oder in einem Installationspaket verwiesen werden kann. Der Anwender muss nicht selber nach den erforderlichen Dateien suchen. Durch ein VERSION-Tag wird sichergestellt, dass die richtige Version einer Komponente benutzt wird. Die Abhängigkeiten von Softwarekomponenten können dabei über mehrere Ebenen ausgedrückt sein, d.h. eine zusätzlich benötigte Komponente kann ebenfalls auf andere benötigte Komponenten verweisen.

OSD dient als allgemeine Grundlage und kann für die Zwecke einzelner Softwarehersteller durch die Verwendung eigener Namensräume (engl. *namespaces*) individuell erweitert werden. Microsoft liefert mit dem Microsoft Internet Component Download (MSICD) ein Beispiel, wie durch die Benutzung einer zusätzlichen Document Type Definition (DTD) die OSD erweiterbar ist. Mit MSCID können dann zum Beispiel neben Java- auch C++- oder Basic-Komponenten beschrieben werden.

Beispiel einer OSD-Datei Es handelt sich also insgesamt um ein Beschreibungskonzept das die *Umgebung* einer Komponente betont, weniger die Komponente selbst. Zur Illustration ist im folgenden eine Windows-Komponente mit dem Namen »Solitaire« (ein Spiel) beschrieben. Erkennbar ist, dass zwei Implementationen vorhanden sind. Die eine Implementation läuft auf Microsoft Betriebssyste-

men und ist unter *http://www.foobar.org/solitaire.cab* zu finden. Die zweite Implementation beruht auf Java und ist in der Datei `www.foo.org/solitaire.jar` zu finden. Für die Java-Anwendung gibt es eine Abhängigkeit (<dependency>). Die angegebene OSD-Datei mit der URL *http://www.foobar.org/cards.osd*, muss ausgewertet werden, damit alle benötigten Komponenten auf dem Client installiert sind.

***Listing 6.22**: Beispiel einer OSD-Datei*

```
<SOFTPKG NAME="com.foobar.www.Solitaire" VERSION="1,0,0,0">
<TITLE>Solitaire</TITLE>
<ABSTRACT>Solitaire by FooBar Corporation</ABSTRACT>
<LICENSE HREF="http://www.foobar.com/solitaire/license.html" />
<!--FooBar Solitaire is implemented in native code for Win32,
Java code for other platforms -->
<IMPLEMENTATION>
<OS VALUE="WinNT"><OSVERSION VALUE="4,0,0,0"/></OS>
<OS VALUE="Win95"/>
<PROCESSOR VALUE="x86" />
<LANGUAGE VALUE="en" />
<CODEBASE HREF="http://www.foobar.org/solitaire.cab" />
</IMPLEMENTATION>
<IMPLEMENTATION>
<IMPLTYPE VALUE="Java" />
<CODEBASE HREF="http://www.foobar.org/solitaire.jar" />
<!-- The Java implementation needs the DeckOfCards object -->
<DEPENDENCY>
<CODEBASE HREF="http://www.foobar.org/cards.osd"/>
</DEPENDENCY>
</IMPLEMENTATION>
</SOFTPKG>
```

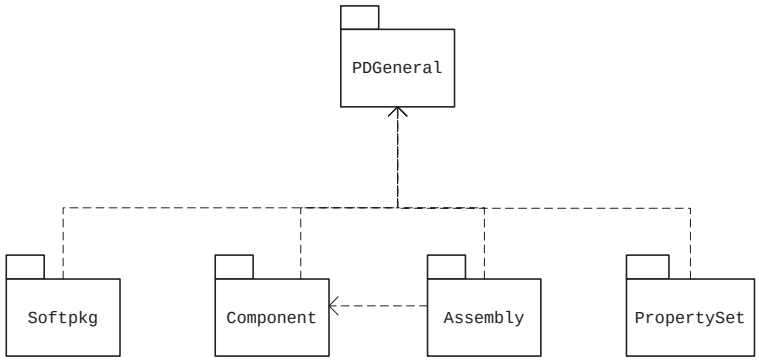
CORBA Component Model Die Object Management Group (OMG) hat mit CORBA eine Architektur zur Unterstüzung von verteilten Objektsystemen etabliert. Ein wichtiger Bestandteil der neuen CORBA 3 Spezifikation ist die Einführung eines expliziten Komponentenmodells, dass das ursprüngliche Obejektkonzept ergänzen soll. Es soll hier nicht weiter auf das CORBA Component Model (CCM) ([OMG, 1999]) selbst eingegangen werden[8]. Der Fokus für diese Arbeit liegt in der im CCM vorgesehen Beschreibung von Komponenten durch sogenannte Deskriptoren, die wiederum unabhängig von der zu Grunde gelegten Komponententechnologie sind. Es besteht dabei ein Komponentenmodell-übergreifender Anspruch, auch wenn sich die Deskriptoren an der Beschreibung von Suns Enterprise JavaBeans, von denen auch das Grundkonzept der CORBA-Komponenten abgeleitet ist, orientieren. Die Austauschbarkeit

[8] Weitere Informationen zu CORBA 3 und dem CORBA Component Modell findet sich auf der Internetseite der OMG `www.omg.org`

von Komponenten auch unterschiedlicher Standards soll somit unterstützt werden. Möglich wird dies trotz des derzeit immer noch fehlenden Konsens über einen Komponentenbegriff zum einen dadurch, dass zumindest einige wesentliche auch allgemein akzeptierte Merkmale für die Komponentensicht existieren (vgl. a. Kapitel 2 und [Griffel, 1998]), zum anderen durch die Wahl einer flexibel erweiterbaren Repräsentation der Beschreibungen.

Die OMG hat zur Beschreibung ihres Meta-Informationsmodells für Komponenten die oben vorgestellte OSD als Grundlage genommen und um einige Aspekte erweitert, so nicht nur die Umgebung sondern auch die Komponenten selbst semantisch reicher beschrieben werden. Es werden also auch hier XML-Dokumente zur Beschreibung einzelner Komponenten, ihrer Kopplungen und des resultierenden Gesamtsystems verwendet. Das zu Grunde liegende Meta-Modell kann auf eine DTD abgebildet werden, sodass einzelne konkrete Modelle gegen das Meta-Modell auf entsprechende Konsistenz getestet werden können. Insbesondere sollen solche Bechreibungen dann in Entwicklungswerkzeugen ausgewertet werden und das Zusammensetzen von Komponenten zu Anwendungen unterstützen.

Abbildung 6.4
Die Bestandteile des CCM-Meta-Informationsmodells



Die Komponentenspezifikationen bestehen dabei aus mehreren Deskriptoren in Form einzelner Dateien, die gemeinsam einen Bestandteil eines Komponentenarchivs, d.h. der Auslieferungsform einer Komponente, bilden. Der erste Deskriptor (*softpkg*) dient der Beschreibung gesamter Softwarepakete und ähnelt in Umfang und Vokabular der OSD. Es handelt sich dabei um allgemeine Angaben wie den Autor oder der Version. Der *component*-Deskriptor dient der Beschreibung der Eigenschaf-

ten der Komponente, die beim Einsatz und bei der Komposition von Komponenten berücksichtigt werden sollen. Hierbei wird die (Oberflächen-) Struktur der Komponente mit ihren Ein- / Ausgängen beschrieben, die bei der Anwendungskomposition verküpft werden. Zu den vorgesehenen Verknüpfungspunkten einer Komponente gehören ebenso Angaben über mögliche zu empfangende oder zu sendende Ereignisse. Ausserdem beinhaltet dieser Deskriptor Informationen über die Art der Umgebung (des *Containers*), in der eine Komponente installiert werden sollte. Der *component*-Deskriptor beinhaltet dabei einen Teil der Informationen, die ansonsten im Rahmen einer CORBA-Anwendung im zugehörigen Interface Repository stehen, sodass nicht unnötig solche externen Informationsquellen abgefragt werden müssen, sondern wie gewünscht möglichst viel Information die Komponente eng begleitet.

Der dritte spezifizierte Deskriptor, das *PropertySet*, beschreibt die Konfiguration einer Komponenten*instanz*, d.h. es können einzelnen Komponentenausprägungen Einstellungen ihrer individuellen Eigenschaften mitgegeben werden. Komponenten können zu grösseren Strukturen bzw. ganzen Anwendungen zusammengefasst werden, was durch einen *Assembly*-Deskriptor beschrieben wird. Es wird dabei jeweils auf die Schnittstellen der einzelnen Bausteine verwiesen. Der *assembly*-Deskriptor stellt auch der späteren Laufzeitumgebung Informationen zum koordinierten Einsatz aller zum Gesamtsystem gehörigen Komponenten bereit. Insgesamt bieten diese erweiterten Bescheibungen eine interessante Grundlage für eine (automatisierte) Konstruktion, wie sie diese Arbeit zum Gegenstand hat. Allerdings unterstützen die CCM-Deskriptoren ausser simplen Namensvergleichen keine Klassifiaktion von Komponenten im Sinne einer Typisierung, sodass sie im Folgenden zwar als Grundlage der Komponentenbeschreibungen eingesetzt werden, allerdings auch entsprechend erweitert werden.

6.2.1.3 Erweiterung der CCM-Deskriptoren

Die semantischen Metainformationen über das Interaktionsmuster einer Komponente sind zunächst durch den `CorbaComponentDescriptor` repräsentiert. Die zu diesem XML-Deskriptor korrespondierende DTD – eine Erweiterung (vgl. Anhang A) der `corbacomponent.dtd` des CCM – liefert dabei die emittierten sowie konsumierten Domänenaktivitäten (»Business Event«) des Komponententyps mit deren syntaktischer Realisierung.

***Listing 6.23**: CorbaComponentDescriptor der Bank*

```
<corbacomponent displayname="BankImpl" componenttypename="Bank">
   <interface name="IDL:CommunicatorInterface:1.0" repid="I1"></interface>
   <interface name="IDL:BankInterface:1.0" repid="I2"></interface>
   <componentfeatures name="Bank" repid="F1">
      <ports>
         <uses usesname ="Komponentenschnittstelle" repid="I1"></uses>
         <provides
           providesname="Komponentenschnittstelle" repid="I1">
         </provides>
         <provides
           providesname="Bankschnittstelle" repid="I2">
         </provides>
         <consumes
           consumesname="Kundenzugang" eventtype="Register"
           instance="Register1">
         </consumes>
         <consumes
           consumesname="Kundenabgang" eventtype="Unregister"
           instance="Unregister1">
         </consumes>
         <consumes
           consumesname="Ueberweisungsauftrag" eventtype="TransferRequest"
           instance="TransferRequest1">
         </consumes>
      </ports>
   </componentfeatures>
</corbacomponent>
```

In Listing 6.23 wird die Bank-Komponente durch die konsumierten (consumes) Business Events Register (Kundenzugang), Unregister (Kundenabgang) sowie TransferRequest (Überweisungsauftrag) charakterisiert. Als *Display Name* wird die Bezeichnung der Komponentenimplementation BankImpl angegeben. Zudem spezifiziert der XML-Deskriptor die angebotenen (provides) und vorausgesetzten (uses) Komponentenschnittstellen, deren Beschreibung hier optional vorgenommen werden kann.

An dieser Stelle muss das Komponenten-Meta-Modell schließlich um zusätzliche DTDs erweitert werden, auf deren Basis nachfolgend der SyntBETypeDescriptor sowie der SyntMessageTypeDescriptor festgelegt werden können. Der SyntBETypeDescriptor erlaubt dabei die Beschreibung von semantischen Nachrichtentypen, die sich aus der Datentypmenge der Parameter sowie dem Datentyp des Rückgabewerts zusammensetzen.

***Listing 6.24**: SyntBETypeDescriptor von TransferRequest1*

```
<syntBEType typename="TransferRequest1">
   <semMessageTypes>
      <semMessageType typename="pruefeKunde" instance="pruefeKunde1">
         <parameter>
            <datatype type="CommunicatorInterface"></datatype>
```

```
         </parameter>
         <returnvalue>
            <datatype type="java.lang.Object"></datatype>
         </returnvalue>
      </semMessageType>
      <semMessageType typename="ueberweise" instance="ueberweise1">
         <parameter>
            <datatype type="CommunicatorInterface"></datatype>
            <datatype type="CommunicatorInterface"></datatype>
            <datatype type="java.lang.Double"></datatype>
         </parameter>
         <returnvalue></returnvalue>
      </semMessageType>
   </semMessageTypes>
</syntBEType>
```

Für den syntaktischen Business-Event-Typ `TransferRequest1` werden in obigem Listing zwei semantische Nachrichtentypen `prüfeKunde` und `überweise` definiert, die jeweils wiederum ihre syntaktische Realisierung referenzieren.

Der `SyntMessageTypeDescriptor` ermöglicht schließlich die Definition von Signaturen – den sogenannten Operationstypen. Zudem werden die Attributwerte für den Kommunikationstyp sowie den Kommunikationsmodelltyp festgelegt.

***Listing 6.25**: `SyntMessageTypeDescriptor` von `überweise1`*

```
<syntMessageType typename="ueberweise1">
   <OperationTypes>
      <OperationType typename="debit">
         <parameter>
            <datatype type="CommunicatorInterface"></datatype>
            <datatype type="CommunicatorInterface"></datatype>
            <datatype type="java.lang.Double"></datatype>
         </parameter>
         <returnvalue></returnvalue>
      </OperationType>
      <OperationType typename="deposit">
         <parameter>
            <datatype type="CommunicatorInterface"></datatype>
            <datatype type="CommunicatorInterface"></datatype>
            <datatype type="java.lang.Double"></datatype>
         </parameter>
         <returnvalue></returnvalue>
      </OperationType>
   </OperationTypes>
   <CommunicationType value="Synchron"></CommunicationType>
   <CommunicationModelType value="MethodCallModel"></CommunicationModelType>
</syntMessageType>
```

Der syntaktische Nachrichtentyp `überweise1` definiert in diesem Beispiel zwei Operationstypen `debit` und `deposit`, die Signaturen – von in diesem Fall entsprechenden Methoden – beschreiben.

Für eine vollständige Typbeschreibung müssen letztendlich noch zwei weitere DTDs definiert werden, die zum einen das

Meta-Modell der Merkmalsstrukturen, zum anderen das der Protokolle repräsentieren. Die Merkmalsstruktur einer Domänenaktivität wird durch den `SemBETypeDescriptor` beschrieben.

***Listing 6.26**: `SemBETypeDescriptor` von `TransferRequest`*

```
<semBEType typename = "TransferRequest">
   <feature name = "TRANSFER" datatype = "tuple_of_4">
      <feature name = "EMPLOYER" datatype = "tuple_of_3">
         <feature name = "NAME" datatype = "String"></feature>
         <feature name = "ACCOUNT_NUMBER" datatype = "Long"></feature>
         <feature name = "BANK_CODE_NUMBER" datatype = "Long"></feature>
      </feature>
      <feature name = "RECEIVER" datatype = "tuple_of_3">
         <feature name = "NAME" datatype = "String"></feature>
         <feature name = "ACCOUNT_NUMBER" datatype = "Long"></feature>
         <feature name = "BANK_CODE_NUMBER" datatype = "Long"></feature>
      </feature>
      <feature name = "PURPOSE" datatype = "String"></feature>
      <feature name = "AMOUNT" datatype = "Double"></feature>
   </feature>
</semBEType>
```

Der in obigem Listing angegebene XML-Deskriptor beschreibt die Merkmalsstruktur zur semantischen Festlegung des Business Events `TransferRequest`. Ein XML-Deskriptor zur Protokollspezifikation auf Basis einer letzten, noch festzulegenden DTD ist nachfolgend dargestellt. Im Beispiel handelt es sich um ein Protokoll über Business Events.

***Listing 6.27**: `BEProtocolDescriptor` der `Bank`*

```
<NFA reference="Bank" name="BankDFA">
   <states>
      <state name="z0" id="z0"></state>
      <state name="z1" id="z1"></state>
   </states>
   <alphabet>
      <element name="Register" id="Register"></element>
      <element name="TransferRequest" id="TransferRequest"></element>
      <element name="Unregister" id="Unregister"></element>
   </alphabet>
   <edges>
      <edge prestate="z0" element="Register" poststate="z1"></edge>
      <edge prestate="z1" element="TransferRequest" poststate="z1"></edge>
      <edge prestate="z1" element="Unregister" poststate="z0"></edge>
   </edges>
   <start id="z0"></start>
   <ends>
      <end id="z0"></end>
   </ends>
</NFA>
```

Der XML-Deskriptor `BEProtocolDescriptor` beschreibt das der semantischen Ebene zu Grunde liegende Protokoll der `Bank` in Form eines endlichen Automaten als 5-Tupel (Zustände,

Alphabet, Kanten, Anfangszustand, Endzustände). Determinismus kann dabei nicht durch die korrespondierende DTD sichergestellt werden, sondern muss in den Deskriptoren bei deren Erstellung explizit beachtet bzw. über ein entsprechendes Werkzeug (vgl. Abschnitt 6.4.2) sichergestellt werden.

Erstellung des Komponentenarchivs Aus der Gesamtheit aller XML-Deskriptoren der *Bank*-Komponente ergibt sich dann ein Komponentenarchiv[9], das mit dem `jar`-Tool des Java Development Kit durch das folgende Kommando erstellt werden kann.

```
jar cvfm Bank.car Bank.mf *.xml
```

Im nachfolgendem Listing ist ein Ausschnitt des Manifests (vgl. a. [Griffel, 1998]) `Bank.mf` – dem »Inhaltsverzeichnis« des Komponentenarchivs `Bank.car` – dargestellt, das von der Typmanagement-Komponente zum Registrierungszeitpunkt eingelesen wird. Im Manifest wird dabei für jedes XML-Dokument der korrespondierende XML-Deskriptor explizit angegeben.

***Listing 6.28**: Manifest des Komponentenarchivs der* Bank

```
Name: Bank.xml
CorbaComponentDescriptor: True

Name: Register.xml
SemBETypeDescriptor: True

Name: TransferRequest.xml
SemBETypeDescriptor: True

Name: Unregister.xml
SemBETypeDescriptor: True

Name: BankDFA.xml
BEProtocolDescriptor: True

Name: CustomerRole.xml
CorbaComponentDescriptor: True

Name: Register_.xml
SemBETypeDescriptor: True

Name: TransferRequest_.xml
SemBETypeDescriptor: True

Name: Unregister_.xml
SemBETypeDescriptor: True

Name: CustomerRoleDFA.xml
BEProtocolDescriptor: True
```

[9] ein »car-Archiv« gemäß der CorbaComponent-Spezifikation

```
Name: Register1.xml
SyntBETypeDescriptor: True

Name: Register1DFA.xml
MessageProtocolDescriptor: True

Name: TransferRequest1.xml
SyntBETypeDescriptor: True

Name: TransferRequest1DFA.xml
MessageProtocolDescriptor: True

Name: Unregister1.xml
SyntBETypeDescriptor: True

Name: Unregister1DFA.xml
MessageProtocolDescriptor: True


Name: registriere1.xml
SyntMessageTypeDescriptor: True

Name: registriere1DFA.xml
OperationProtocolDescriptor: True

Name: pruefeKunde1.xml
SyntMessageTypeDescriptor: True

Name: pruefeKunde1DFA.xml
OperationProtocolDescriptor: True

Name: ueberweise1.xml
SyntMessageTypeDescriptor: True

Name: ueberweise1DFA.xml
OperationProtocolDescriptor: True

Name: deregistriere1.xml
SyntMessageTypeDescriptor: True

Name: deregistriere1DFA.xml
OperationProtocolDescriptor: True
```

Auf Basis dieses Komponentenarchivs kann der XML-Parser die Abbildung der XML-Beschreibungen auf den Typmanager-internen Typbaum vornehmen.

6.2.1.4 Parsing-Phase

Nach dem Einlesen des Komponentenarchivs wird zunächst für jeden XML-Deskriptor das Document Object Model (DOM) mit Hilfe des XML-Parsers aufgebaut. Dieses ist im Anschluss daran auf den Typmanager-internen Typbaum abzubilden. Dabei wird zu den entsprechenden XML-Deskriptoren jeweils eine korrespondierende Java-Klasse implementiert, die nach ihrer Instanziierung einen Teilbereich des aufgebauten internen (Java-) Typbaums darstellt. Dazu ist zunächst eine abstrakte Basisklas-

se TypehierarchyElement implementiert, von der nachfolgend die Java-Klassen zur Repräsentation der konkreten Elemente des Typbaums abgeleitet werden können. Die Klasse TypehierarchyElement ermöglicht dabei den Aufbau einer hierarchischen Baumstruktur. Sie besitzt daher neben dem Namensattribut als weiteres Attribut eine Menge, die Referenzen auf alle Subelemente beinhaltet.

6.2.1.5 Abbildung der XML-Deskriptoren

Der CorbaComponentDescriptor wird in der internen Struktur des Komponententyps auf das Wurzel-Element der semantischen Ebene abgebildet – den *Komponententyp*, zu dem die Java-Klasse ComponentType korrespondiert. Des weiteren werden XML-Deskriptoren benötigt, die die den Business Events zu Grunde liegende Semantik in Form von Merkmalsstrukturen (SemBETypeDescriptor) sowie deren mögliche Sequenzen – die Protokolle über Business Events (BEProtocolDescriptor) – beschreiben. Optional können zudem Constraints angegeben werden (ConstraintDescriptor). Nach dem Parsen der entsprechenden XML-Deskriptoren kann schließlich ein Objekt der Klasse ComponentType instanziiert werden, wobei die eingelesenen Metainformationen dem Konstruktor als Parameter übergeben werden.

```
public class ComponentType extends TypehierarchyElement
{
   // Attribute
   private String componentTypeNameM;
   private Set firedEventsM;
   private Set consumedEventsM;
   private Set protocolsOfEventsM;
   private Set constraintsM;

   public ComponentType(String typeName, Set firedBEs, Set consumedBEs,
   Set protocols, Set constraints)
   {
      super(typeName);
      super.addSubelement(firedBEs);
      super.addSubelement(consumedBEs);
      super.addSubelement(protocols);
      super.addSubelement(constraints);

      componentTypeNameM = typeName;
      firedEventsM = firedBEs;
      consumedEventsM = consumedBEs;
      protocolsOfEventsM = protocols;
      constraintsM = constraints;

      ...
   }

   ...
}
```

Java-Klasse ComponentType

Im Konstruktor der Klasse ComponentType werden der Superklasse TypehierarchyElement der Typname typeName sowie alle untergeordneten Elemente übergeben. Dies sind die Mengen der emittierten und konsumierten Business Events firedBEs und consumedBEs, die Protokollmenge protocols mit den Protokollen über Business Events sowie die Constraint-Menge constraints.

Der SyntBETypeDescriptor wird im internen Typbaum auf das Wurzel-Element der Nachrichtenebene abgebildet – den *syntaktischen Business-Event-Typ*, dem die Java-Klasse SyntBEType zugeordnet ist. Auch hier wird wiederum ein XML-Deskriptor für die Protokollbeschreibung benötigt (MessageProtocolDescriptor), sodass die gesamten Metainformationen zur Instanziierung der Klasse SyntBEType dem Konstruktor als Parameter übergeben werden können.

```
public class SyntBEType extends TypehierarchyElement
{
   // Attribute
   private String syntBETypeNameM;
   private Set semMessageTypesM;
   private Set protocolsOfMessagesM;

   // Referenz auf übergeordneten Komponententyp (Default)
   private ComponentType ctM = null;

   public SyntBEType(String typeName, Set semMessageTypes,
   Set protocols)
   {
      super(typeName);

      super.addSubelement(semMessageTypes);
      super.addSubelement(protocols);

      syntBETypeNameM = typeName;
      semMessageTypesM = semMessageTypes;
      protocolsOfMessagesM = protocols;

      ...
   }

   ...
}
```

Java-Klasse SyntBEType

Im Konstruktor der Klasse SyntBEType werden der Superklasse TypehierarchyElement der Typname typeName sowie alle untergeordneten Elemente übergeben. Dies sind in diesem Fall die Menge der semantischen Nachrichtentypen semMessageTypes sowie die Protokollmenge protocols, die Protokolle über die entsprechenden Nachrichten beinhaltet.

Der SyntMessageTypeDescriptor wird in der internen Struktur des Komponententyps auf das Wurzel-Element der syntaktischen Ebene abgebildet – den *syntaktischen Nachrichtentyp*, dem die Java-Klasse SyntMessageType entspricht. An dieser Stelle ist abermals ein XML-Deskriptor für die Protokollbeschreibung vorzusehen (OperationProtocolDescriptor). Nach dem Parsen der relevanten XML-Deskriptoren kann somit ein Objekt der Klasse SyntMessageType instanziiert werden, wobei die eingelesenen Metainformationen dem Konstruktor wiederum als Parameter übergeben werden.

```
public class SyntMessageType extends TypehierarchyElement
{
   // Attribute
   private String syntMessageTypeNameM;
   private Set operationTypesM;
   private Set protocolsOfOperationsM;
   private CommunicationType communicationTypeM;
   private CommunicationModelType communicationModelTypeM;

   // Referenz auf übergeordneten synt. BE-Typ
   private SyntBEType sbetM = null;

   public SyntMessageType(String typeName, Set operationTypes,
   Set protocols, CommunicationType ct, CommunicationModelType cmt)
   {
      super(typeName);

      super.addSubelement(operationTypes);
      super.addSubelement(protocols);
      super.addSubelement(ct);
      super.addSubelement(cmt);

      syntMessageTypeNameM = typeName;
      operationTypesM = operationTypes;
      protocolsOfOperationsM = protocols;
      communicationTypeM = ct;
      communicationModelTypeM = cmt;

      ...
   }
   ...
}
```

Java-Klasse SyntMessageType

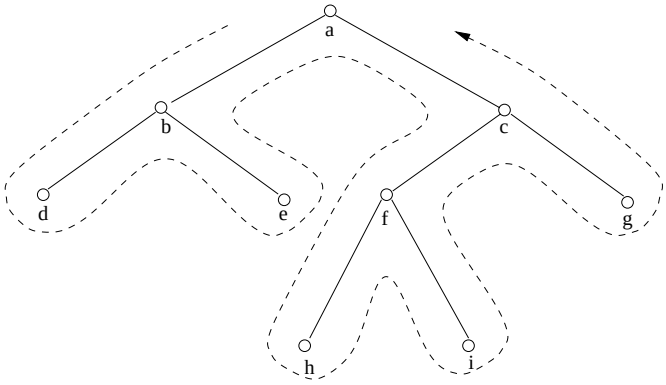
Im Konstruktor der Klasse SyntMessageType werden der Superklasse TypehierarchyElement der Typname typeName sowie alle untergeordneten Elemente übergeben. Dies sind hier die Menge der Operationstypen operationTypes wie auch deren Protokollmenge protocols. Weitere Elemente stellen die Attribute Kommunikationstyp ct und Kommunikationsmodelltyp cmt dar. Auf das vollständige Klassendiagramm des Typmodells soll an dieser Stelle nicht weiter eingegangen werden (s. Anhang E).

Die Abbildung des durch den XML-Parser aufgebauten Document Object Model (DOM) auf die interne Struktur des Komponententyps (Typbaum) soll an dieser Stelle nur am Beispiel der Merkmalsstrukturen betrachtet werden, da sich die Notwendigkeit der Implementation eines rekursiven Abbildungsalgorithmus ergibt. Dieser wird im Folgenden dargestellt. Alle anderen Elemente des Typbaums lassen sich 1:1 vom Document Object Model auf die Typmanager-interne Repräsentation abbilden, sodass hierzu keine weiteren (komplexen) Abbildungsalgorithmen betrachtet werden müssen.

6.2.1.6 Abbildungsalgorithmus für Merkmalsstrukturen

Für Merkmalsstrukturen ergibt sich das Problem der Abbildung des durch den XML-Parser aufgebauten Document Object Model auf den internen Typbaum. Die Schwierigkeit der Abbildung besteht insbesondere darin, dass Merkmalsstrukturen beliebige Baumstrukturen zu Grunde liegen können, wobei die DOM-Repräsentation im ersten Schritt mittels Tiefensuche relativ effizient durch einen rekursiven Algorithmus durchlaufen werden kann.

Abbildung 6.5
Baumdurchlauf bei Tiefensuche



Auf der Basis der Tiefensuche resultiert für das obige Beispiel die folgende Knotenreihenfolge: *a, b, d, e, c, f, h, i, g*

In diesem Zusammenhang ergibt sich allerdings das Problem, dass Merkmalsstrukturen nach dem *bottom-up*-Prinzip aufgebaut werden müssen, sodass elementare Konzepte und Typen in einem ersten Schritt erstellt werden, aus denen sich schließlich zusammengesetzte Konzepte sowie Typen ableiten lassen. Ein Algorithmus, der die Tiefensuche implementiert, muss dadurch mit einer Zwischenspeicherung arbeiten, da zusammengesetzte

Konzepte und Typen erst dann vollständig definiert sind, wenn alle untergeordneten Knoten ausgewertet sind. Für diese Zwischenspeicherung werden Listen verwendet, in denen Konzepte und Merkmalsmengen abgelegt werden, die erst zur weiteren Laufzeit komplettiert werden können. Zudem wird die Anzahl noch benötigter Subkonzepte beziehungsweise Merkmale für das Superkonzept in einer weiteren Liste zwischengespeichert. Diese Zwischenspeicherung ist dabei für alle Nicht-Blatt-Knoten relevant.

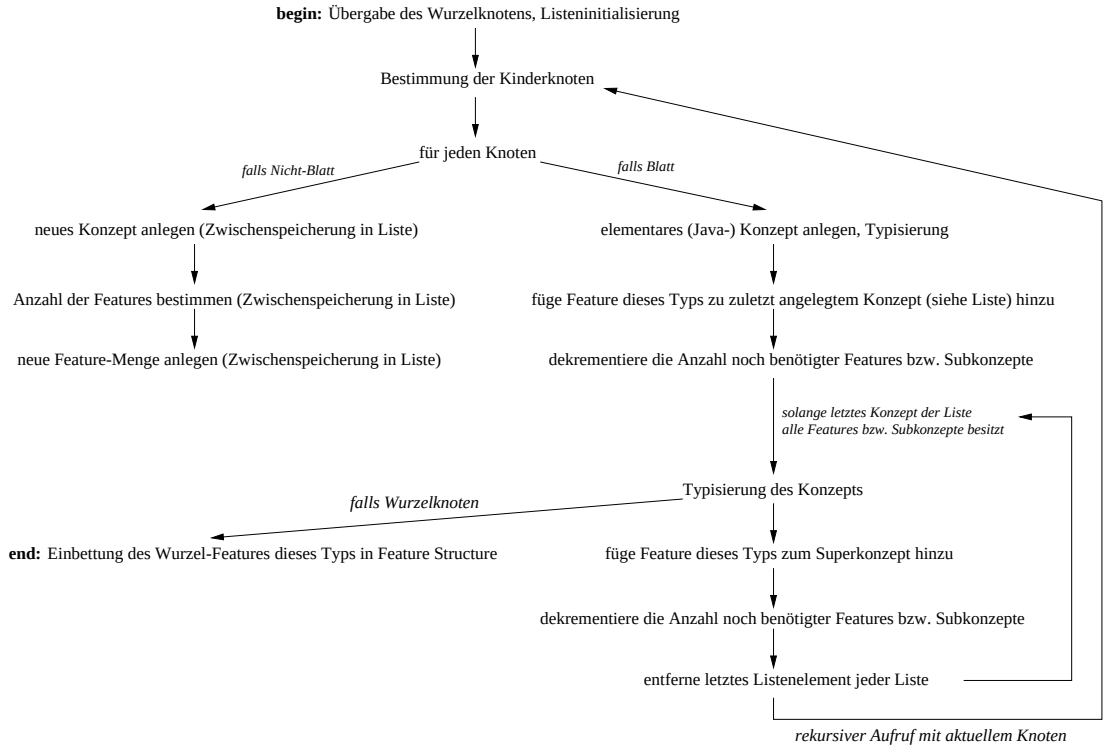


Abbildung 6.6
Ablauf des Abbildungsalgorithmus

Für die beschriebene Typregistrierung kann eine Methode über das Application Programming Interface (API) der Typmanagement-Komponente zur Verfügung gestellt werden.

```
   public void register(String filename, boolean adapter,
   String domainpath);

Parameter:
   filename - Name des Komponentenarchivs
   adapter - Kennzeichnung als Adapterkomponente mit »true«
   domainpath - Pfad der relevanten Typdomäne
```

Über die grafische Benutzeroberfläche kann ebenso die Registrierung eines Komponententyps vorgenommen werden, indem über den *Type Manager Explorer* zunächst eine entsprechende Typdomäne selektiert und über den Menüpunkt »Administration« der Untermenüpunkt »Register Component Type« aufgerufen wird. Im letzten Schritt muss jetzt lediglich das Komponentenarchiv der zu registrierenden Komponente im *Register Component Type Dialog* ausgewählt werden.

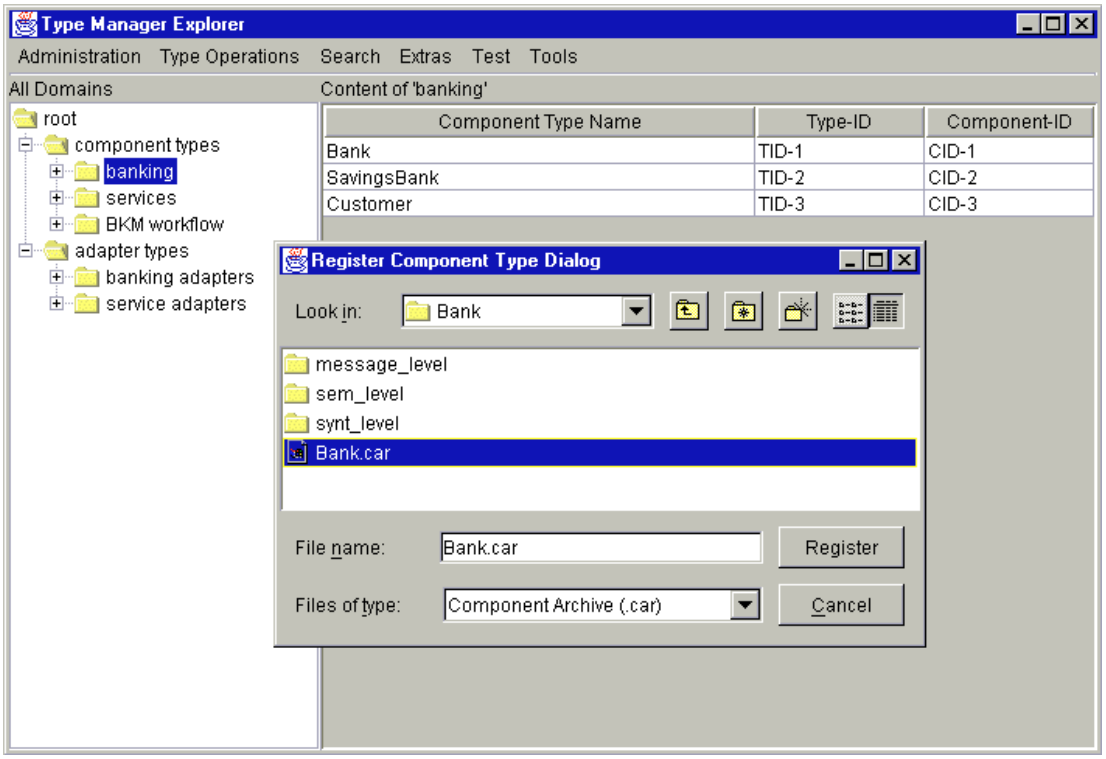


Abbildung 6.7
Typregistrierung im Type Manager Explorer

Im Anschluss an die Typregistrierung sollen im weiteren Verlauf der Implementationsdokumentation die auf dem Typmodell definierten Typoperationen betrachtet werden. Schwerpunkt der folgenden Betrachtung sind der *Typvergleich* sowie die *Kompatibilitätsprüfung*, die jeweils im Anschluss an eine Typregistrierung automatisch von der Typmanagement-Komponente auf der Grundlage aller momentan in der Domäne registrierten Komponententypen durchgeführt werden. Durch die dabei evaluierten Assoziationen wird eine Einordnung des Komponententyps in die Gesamthierarchie aller Komponententypen vorgenommen und somit eine Erweiterung der Wissensbasis vollzogen.

6.2.2 Typoperationen

Nach der Registrierung entsprechender Komponententypen können Operationen auf den Typrepräsentationen durchgeführt werden. Der Fokus liegt dabei insbesondere auf dem Typvergleich sowie der Kompatibilitätsprüfung, die anhand ihrer zu Grunde liegenden Algorithmen vorgestellt werden. Während sich der *Typvergleich* auf die Ableitung von verschiedenen Beziehungen zwischen Komponententypen stützt (Gleichheit, Typarten, Basistypen), basiert die *Kompatibilitätsprüfung* lediglich auf Kompatibilitätsbeziehungen, die die Interaktions- beziehungsweise Kopplungsfähigkeit der Komponententypen reflektieren. Allerdings kann im Rahmen eines Typvergleichs ebenso eine Aussage über die Konformität des zu Grunde liegenden Substitutionskandidaten abgeleitet werden.

6.2.2.1 Typvergleich

Im Folgenden werden die wesentlichen Code-Fragmente eines Typvergleichsalgorithmus vorgestellt, der den Vergleich zweier Komponententypen realisiert.

***Listing 6.29**: Listingauszug: Typvergleich*

```
public Report compare(ComponentType ct1, ComponentType ct2)
{
  Report report = new Report(...);
  ...

  // Initialisierung
  boolean semType = false; // Konformität des sem. Typs
  boolean approxSyntType = false; // Konformität des approx. synt. Typs}
  boolean syntTypeResult = false; // Konformität des synt. Typs

  // Festhalten des besten Konformitätsergebnisses für jedes Protokoll über
  // BEs, Nachrichten
  Hashtable protocolsOfBEs = new Hashtable();
  Hashtable protocolsOfMessages = new Hashtable();

  boolean semRelation = false;

  // Objektregistrierung zum sem. Typvergleich
  ComponentType.registerComponentType(ct1, ct2, ...);

  if(ComponentType.isEqual()) // Prüfung auf Gleichheit
  {
    // Assoziationen zwischen Komponententypen ziehen}
    ct1.equals.add(ct2);
    ct2.equals.add(ct1);

    semType = true; // Typgleichheit -> Substitutionskandidat
    semRelation = true;
```

```
}
if(ComponentType.isTypekind1()) // Prüfung auf Typart-1
...

// Kurzaussage des Reports (Initialisierung)
boolean[] summary = new boolean[3];
summary[0] = semType;
summary[1] = approxSyntType;
summary[2] = syntTypeResult;

if(semRelation) // falls eine Beziehung auf sem. Ebene
{
  // für gleiche BE-Protokolle
 if(BEProtocolTester.isEqual())
 {
   // einander entsprechende Zustandsübergänge beider BE-Protokolle
   Vector pairsOfSemBEs = BEProtocolTester.getPairs();
   approxSyntType = true;
   syntTypeResult = true;

   for(int i=0 ; i < pairsOfSemBEs.size() ; i++)
   {
     // Objektregistrierung zum Vergleich des synt. BE-Typs
     SyntBEType.registerSyntBEType(sbet1, sbet2, ...);
     boolean approxRelation = false;

     if(SyntBEType.isEqual()) // Prüfung auf Gleichheit}
     {
       // Assoziationen zwischen synt. BE-Typen ziehen}
       sbet1.equals.add(sbet2);
       sbet2.equals.add(sbet1);
       approxRelation = true;
     }
     if(SyntBEType.isTypekind1()) // Prüfung auf Typart-1
     ...

     if(!approxRelation)}
       approxSyntType = false;
     else // falls eine Beziehung auf Nachrichtenebene
     {
       // für gleiche Nachrichtenprotokolle
       if(messageProtocolTester.isEqual() ...)
       {
         // einander entsprechende Zustandsübergänge beider
         // Nachrichtenprotokolle
         Vector pairsOfSemMessages = messageProtocolTester.getPairs();
         boolean syntType = true;

         for(int j=0 ; j < pairsOfSemMessages.size() ; j++)
         {
           // Objektregistrierung zum synt. Typvergleich
           SyntMessageType.registerSyntMessageType(syntMT1, syntMT2, ...);
           boolean syntRelation = false;

           if(SyntMessageType.isEqual()) // Prüfung auf Gleichheit}
           {
             // Assoziationen zwischen synt. Nachrichtentypen ziehen
             syntMT1.equals.add(syntMT2);
```

```
                syntMT2.equals.add(syntMT1);
                syntRelation = true;
              }
              if(SyntMessageType.isTypekind1()) // Prüfung auf Typart-1}
              ...
              if(!syntRelation)
                syntType = false;

          }
        }
        // Festhalten des besten Ergebnisses für das Nachrichtenprotokoll in}
        // "protocolsOfMessages"
        ...
      }
    }

    // konjunktive Verknüpfung der Resultate aller Nachrichtenprotokolle
    syntTypeResult = ...
  }
  // Festhalten des besten Ergebnisses für das BE-Protokoll
  // in "protocolsOfBEs"
  ...

  // konjunktive Verknüpfung der Resultate aller BE-Protokolle
  approxSyntType = ...
  syntTypeResult = ...

  // Regeln
  if(!semType)
    approxSyntType = false;
  if(!approxSyntType)
   syntTypeResult = false;

  summary[1] = approxSyntType;
  summary[2] = syntTypeResult;
  report.setSummary(summary);

  return report;
}
```

Der dargestellte Algorithmus instanziiert zunächst ein Objekt der Klasse `Report`, das eine dynamische Liste (Java-Klasse `Vector`) von Zeichenketten (Java-Klasse `String`) für die Erstellung einer textbasierten Auswertung des Typvergleichs realisiert. Des weiteren beinhaltet dieser Report eine zusammenfassende Kurzaussage `summary` über die Konformität des als Substitutionskandidat spezifizierten Komponententyps `ct1` auf den drei festgelegten Ebenen des Typbaums, wobei die in Abschnitt 4.6.1 angegebenen Typausprägungen als *konform* eingestuft werden. Diese Kurzaussage ist als `Boolean`-Array implementiert. Die Konzeption des Algorithmus sieht dabei vor, für jede abgeleitete Beziehung zwischen den Elementen beider Typbäume eine entsprechende Assoziation zwischen diesen zu etablieren, die programmiersprachlich als Referenz implementiert ist.

In der Initialisierungsphase des Algorithmus wird die Instanziierung zweier Tabellen (Java-Klasse `Hashtable`) vorgenom-

men, in denen die vollständigsten (besten) Konformitätsergebnisse für jedes Protokoll über Business Events beziehungsweise über Nachrichten des zu ersetzenden Komponententyps `ct2` festgehalten werden. Die Tabelle `protocolsOfBEs` speichert dabei zusammenfassend (über alle Zustandsübergänge) für jedes BE-Protokoll das Konformitätsergebnis der untergeordneten zweiten sowie dritten Ebene des Typbaums in Form jeweils eines `Boolean`, während in `protocolsOfMessages` lediglich das Konformitätsergebnis der untergeordneten dritten Ebene durch ein `Boolean` abgelegt wird.

Im Anschluss daran kann mit der Prüfung der durch die festgelegten Assoziationen definierten Beziehungen auf semantischer Ebene (s. Abschnitt 4.4.3.1) für die als Parameter übergebenen Komponententypen `ct1` und `ct2` begonnen werden. Falls eine derartige Relation abgeleitet wurde, können – für den Fall gemeinsamer Protokolle – die einander entsprechenden Zustandsübergänge beider BE-Protokolle `pairsOfSemBEs` bezüglich ihrer syntaktischen Realisierung `sbet1` und `sbet2` einem Typvergleich auf Nachrichtenebene (s. Abschnitt 4.4.3.2) unterzogen werden. Für den Fall einer abgeleiteten Beziehung auf Nachrichtenebene können wiederum – auf der Basis gemeinsamer Protokolle – die syntaktischen Realisierungen `syntMT1` und `syntMT2` einander entsprechender Zustandsübergänge beider Nachrichtenprotokolle `pairsOfSemMessages` verglichen werden, die letztendlich den Typvergleich (nach Abschnitt 4.4.3.3) auf syntaktischer Ebene realisieren.

Das vollständigste (beste) Konformitätsergebnis für das jeweils zu Grunde liegende Protokoll des zu ersetzenden Komponententyps wird nach entsprechender Ableitung der Beziehungen in den Tabellen `protocolsOfMessages` sowie `protocolsOfBEs` festgehalten. Für ein Business Event müssen abschließend die Resultate aller Nachrichtenprotokolle konjunktiv verknüpft werden, um das Gesamtergebnis abzuleiten, das daraufhin in der Tabelle `protocolsOfBEs` abgelegt wird. Vor seiner Terminierung führt der Algorithmus schließlich die Konjunktion aller Resultate der BE-Protokolle durch, setzt die entsprechenden Werte der `summary` und liefert diese durch das korrespondierende `Report`-Objekt zurück.

6.2.2.2 Protokollvergleich

Insbesondere soll an dieser Stelle noch auf den dem Algorithmus zu Grunde liegenden Protokollvergleich eingegangen werden. Der zugrunde gelegte Algorithmus aus Kapitel 3 bzw. des-

sen Implementation aus Abschnitt 6.1, der die Gleichheit zweier DFAs prüft, wird im Folgenden dahingehend modifiziert, dass für die Zustandsübergänge nicht Kantenbeschriftungen in Form von Zeichenketten, sondern die strukturellen Beschreibungen der Business Events, Nachrichten beziehungsweise Operationen herangezogen werden.[10] Unter Gleichheit wird hier Äquivalenz verstanden, d.h., dass beide Automaten die gleiche Sprache akzeptieren beziehungsweise durch die gleiche reguläre Menge beschrieben werden können.

Der Algorithmus wird mit dem Paar der Anfangszustände beider Automaten initialisiert und sucht anschließend mögliche Folgezustandspaare auf der Basis der gegebenen Alphabete. Er terminiert, wenn keine neuen Folgezustandspaare mehr gefunden werden. Bei der Suche nach einer Entsprechung zu einem gegebenen Zustandsübergang im Vergleichsautomat wird der namensbasierte Ansatz (Kantenbeschriftung durch Zeichenkette) wie folgt erweitert:

- suche Strukturbeschreibung zu gegebener Kantenbeschriftung *m*
- suche strukturelle Entsprechung im Vergleichsautomat, hole deren Kantenbeschriftung *m'*
- prüfe, ob vom gegebenen Zustandspaar mit *m* und *m'* ein Folgezustandspaar erreichbar ist

Das API des Typmanagers umfasst dementsprechend eine Methode über die ein Typvergleich angestossen werden kann.

```
   public Report compare(ComponentType ct1, ComponentType ct2);

Parameter:
   ct1 - der Substitutionskandidat
   ct2 - der zu ersetzende Komponententyp
gibt zurück:
   textbasierter Bericht des Typvergleichs mit Kurzaussage
   über Konformität
```

An dieser Stelle ergibt sich die Möglichkeit der Bereitstellung einer Zusatzfunktion, die einen ausschließlich syntaktischen Typvergleich realisiert. Die wesentliche Motivation ist dabei die Abkopplung der semantischen Ebene, was in einigen technischen

[10] Der Ansatz dieser arbeitet definiert Semantik ja gerade über strukturelle Gleichheit, sodass die entsprechenden Stukturen berücksichtigt werden sollten.

Anwendungskontexten durchaus sinnvoll sein kann. Auf dieser Basis können schließlich zwei syntaktische Business-Event-Typen analog zu dem in Abbildung 6.29 vorgestellten Algorithmus verglichen werden. Allerdings kann auf Grund der dann fehlenden Semantik keine Kurzaussage über deren Konformität generiert werden, sodass der `Report` in diesem Fall keine `summary` beinhaltet.

```
   public Report compareSyntBETypes(SyntBEType sbet1, SyntBEType sbet2);

Parameter:
   sbet1 - ein syntaktischer Business-Event-Typ
   sbet2 - ein syntaktischer Business-Event-Typ
gibt zurück:
   textbasierter Bericht des Typvergleichs
```

6.2.3 Kompatibilitätsprüfung

Im Anschluss werden die wesentlichen Code-Fragmente eines Algorithmus zur Kompatibilitätsprüfung im Sinne des Typmanagements zweier Komponententypen vorgestellt.

***Listing 6.30**: Listingauszug: Kompatibilitätsprüfung*

```
public Report checkCompatibility(ComponentType ct1, ComponentType ct2)
{
  Report report = new Report(...);
  ...
  // Prüfung der semantischen Kompatibilität
  boolean isCompatible = ComponentType.isCompatible(ct1, ct2, ...);
  ...

  // Kurzaussage des Reports (Initialisierung)
  boolean[] summary = new boolean[3];
  summary[0] = isCompatible;  // Kompatibilität des sem. Typs
  summary[1] = false; // Kompatibilität des approx. synt. Typs
  summary[2] = false; // Kompatibilität des synt. Typs

  if(isCompatible) // falls semantische Kompatibilität
  {
    // Assoziationen zwischen Komponententypen ziehen
    ct1.compatibleTypes.add(ct2);
    ct2.compatibleTypes.add(ct1);

    // Initialisierung
    boolean approxSyntType = false;
    boolean syntType = false;
    ...

    // für der Kompatibilität zu Grunde liegende BE-Protokolle
    if(BEProtocolTester.isEqual() && ComponentType.checkCompatibility(...))
    {
      // einander entsprechende Zustands"uberg"ange beider BE-Protokolle
     Vector pairsOfSemBEs = BEProtocolTester.getPairs();
```

```
    approxSyntType = true;
    syntType = true;

    for(int i=0 ; i < pairsOfSemBEs.size() ; i++)
    {

    // Kompatibilitätsprüfung des approx. synt. Typs
    boolean compatibleBEs = SyntBEType.isCompatible(sbet1, sbet2, ...);
    ...

    if(!compatibleBEs)
      approxSyntType = false;
    else // falls Kompatibilität auf Nachrichtenebene
    {
      // Assoziationen zwischen synt. BE-Typen ziehen
      sbet1.compatibleTypes.add(sbet2);
      sbet2.compatibleTypes.add(sbet1);
      ...
      // für der Kompatibilität zu Grunde liegende Nachrichtenprotokolle
      if(messageProtocolTester.isEqual())
      {
        // einander entsprechende Zustandsübergänge beider
        // Nachrichtenprotokolle
        Vector pairsOfSemMessages = messageProtocolTester.getPairs();

        for(int j=0 ; j < pairsOfSemMessages.size() ; j++)
        {
          // Prüfung der syntaktischen Kompatibilität
          boolean compatibleMessages =
              SyntMessageType.isCompatible(syntMT1, syntMT2, ...);

          if(compatibleMessages) // falls syntaktische Kompatibilität
          {
            // Assoziationen zwischen synt. Nachrichtentypen ziehen
            syntMT1.compatibleTypes.add(syntMT2);
            syntMT2.compatibleTypes.add(syntMT1);
          }
          else
            syntType = false;
        }
      }
    }
  }
}
// Aufbereitung der Kurzaussage des Reports

// Regel
if(!approxSyntType)
  syntType = false;

// Festhalten des besten Ergebnisses
if(!summary[1])
{
  summary[1] = approxSyntType;
  summary[2] = syntType;
}
else
{
```

```
    if(!summary[2])
      summary[2] = syntType;
  }

  report.setSummary(summary);
  return report;
}
```

Der dargestellte Algorithmus instanziiert im ersten Schritt – analog zum Typvergleich – ein Objekt der Klasse `Report` für die Erstellung einer textbasierten Auswertung der Kompatibilitätsprüfung. Diese beinhaltet eine zusammenfassende Kurzaussage `summary` über die Kompatibilität der Komponententypen auf den drei Ebenen des Typbaums. Die Kurzaussage ist wiederum als `Boolean`-Array realisiert. Die Konzeption des Algorithmus sieht dabei vor, für jede abgeleitete Kompatibilitätsbeziehung zwischen den Elementen beider Typbäume eine entsprechende Assoziation zwischen diesen herzustellen, die programmiersprachlich als Referenz implementiert ist.

Nach einer entsprechenden Initialisierung des Algorithmus wird zunächst die semantische Kompatibilität (s. Abschnitt 4.4.3.1) beider als Parameter übergebener Komponententypen `ct1` und `ct2` überprüft. Diese Beziehung muss dabei nicht unbedingt zwischen den Default-Typen zustande kommen, wenn die Komponenten noch weitere Rollen spezifizieren. Gefordert wird an dieser Stelle mindestens eine Rolle, in der die Komponententypen kompatibel zueinander sind (vgl. a. Abschnitt 4.8). Im Erfolgsfall müssen daraufhin alle der semantischen Kompatibilität zu Grunde liegenden Protokolle über Business Events bezüglich ihres approximierten syntaktischen Typs analysiert werden. Dazu werden die einander entsprechenden Zustandsübergänge beider BE-Protokolle `pairsOfSemBEs` im Hinblick auf ihre korrespondierenden syntaktischen Realisierungen `sbet1` und `sbet2` auf Kompatibilität geprüft. Stellt sich dabei die Kompatibilität des zu Grunde liegenden syntaktischen Business-Event-Typs (s. Abschnitt 4.4.3.2)) heraus, so werden wiederum alle korrespondierenden Protokolle – in diesem Fall solche über die entsprechenden Nachrichten – zur Prüfung der syntaktischen Kompatibilität (s. Abschnitt 4.4.3.3) herangezogen. Die Basis dieser Prüfung stellen abermals die einander entsprechenden Zustandsübergänge `pairsOfSemMessages` beider Nachrichtenprotokolle dar, auf deren syntaktischen Realisierungen `syntMT1` und `syntMT2` schließlich eine syntaktische Kompatibilitätsprüfung durchgeführt wird.

Zur Aufbereitung der Kurzaussage des Berichts muss abschließend für jedes Protokoll über Business Events dessen Kom-

patibilitätsergebnis auf der Nachrichtenebene sowie der syntaktischen Ebene (über alle Zustandsübergänge) zusammenfassend betrachtet werden. Dabei wird in der `summary` stets das Ergebnis für das Protokoll mit dem vollständigsten (besten) Kompatibilitätsergebnis festgehalten. Der Algorithmus terminiert letztendlich mit der Rückgabe des `Report`-Objekts inklusive seiner `summary`.

Für die Kompatibilitätsprüfung zweier Komponententypen steht die folgende API-Methode zur Verfügung:

```
   public Report checkCompatibility(ComponentType ct1,
   ComponentType ct2);

Parameter:
   ct1 - ein Komponententyp
   ct2 - ein Komponententyp
gibt zurück:
   textbasierter Bericht der Kompatibilitätsprüfung mit Kurzaussage
   über Kompatibilität
```

Als Alternative zur Benutzung des API kann an dieser Stelle ebenso auf die grafische Benutzeroberfläche zurückgegriffen werden. Der *Type Manager Explorer* stellt dabei über den Menüpunkt »Type Operations« für den Typvergleich sowie die Kompatibilitätsprüfung die Aufbereitung eines Berichts *Type Comparison Report* beziehungsweise *Type Compatibility Report* zur Verfügung, die jeweils detaillierte Informationen über die evaluierten Beziehungen zwischen allen korrelierenden Elementen der Typbäume beinhalten. Konformitätsbeziehungen (s. konforme Typausprägungen in Abschnitt 4.6.1) im Kontext eines Typvergleichs sowie Kompatibilitätsbeziehungen (s. Abschnitt 4.6.2) im Rahmen einer Kompatibilitätsprüfung werden in ihrer textuellen Form stets grün unterlegt, um diese Information hervorzuheben. Im Gegensatz dazu impliziert eine rote Unterlegung, dass eine derartige Beziehung nicht abgeleitet werden konnte.

Über den Button »Store« kann der Bericht als Textdatei[11] gespeichert werden, während »Reject« den Bericht verwirft. Der Button « Summary« erlaubt die Anzeige der zusammenfassenden Kurzaussage über die *Konformität* des Substitutionskandidaten beziehungsweise die *Kompatibilität* der Interaktionspartner auf den drei Ebenen des Typbaums.

Die durch die Typoperationen etablierten Assoziationen zwischen Typrepräsentationen können nachfolgend als Basis der Funktionen zur Typsuche ausgewertet werden.

[11] Auf diese Weise ist eine automatische Auswertung durch eine spätere Textanalyse denkbar, sodass der Autor diese Möglichkeit bewusst vorgesehen hat.

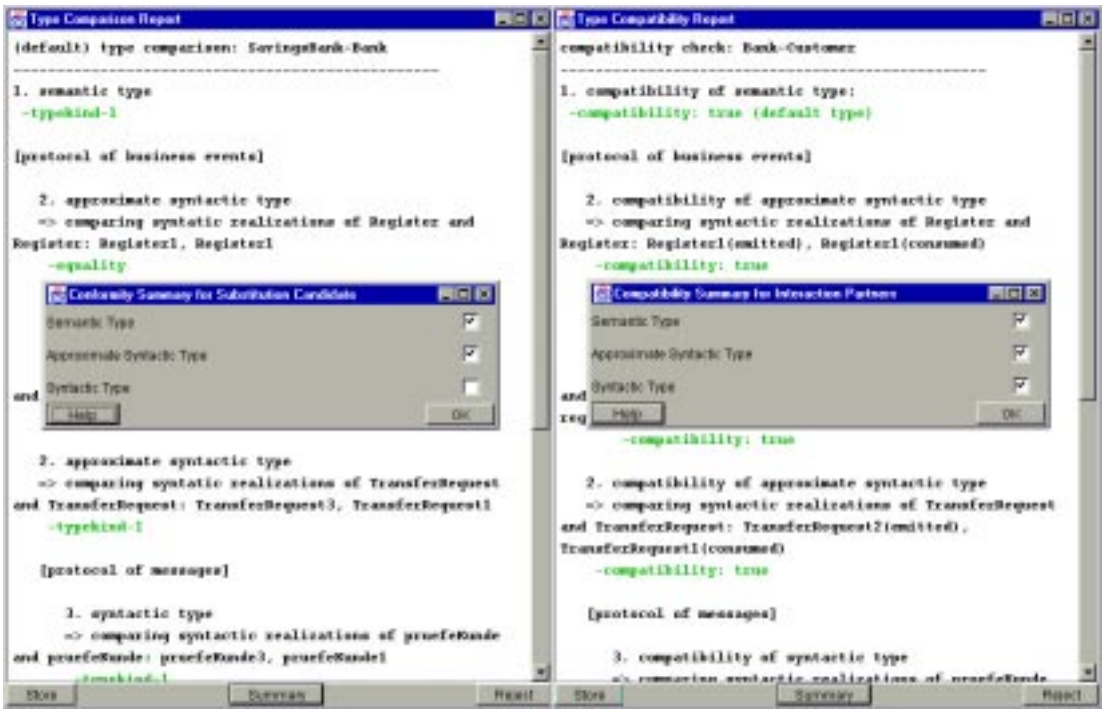


Abbildung 6.8
Typoperationen des Type Manager Explorer

6.2.4 Typsuche

Für die Typsuche sind einerseits die durch den Typvergleich evaluierten Assoziationen relevant, auf die sich die Suche nach Basistypen und Typarten stützt, sowie andererseits die Kompatibilitäts-Assoziationen, auf deren Basis potenzielle Interaktionspartner – also kompatible Komponententypen – gesucht werden können. Als Zusatzfunktion steht letztendlich eine Funktion zur Adaptersuche zur Verfügung, mit deren Hilfe potenzielle Adapterkomponenten gefunden werden können, die gemäß Abschnitt 6.1 zwischen zwei zu koppelnde, zunächst inkompatible Komponenten geschaltet werden können, um deren Interaktion zu ermöglichen.

6.2.4.1 Basistypen und Typarten

Auf der Grundlage von evaluierten Beziehungen zwischen Typbäumen durch entsprechende Typvergleiche, für die Assoziationen etabliert wurden, können Basistypen sowie Typarten zu einem ausgewählten Komponententyp gesucht werden. Der Suchvorgang stützt sich dabei stets auf die dem selektierten Kompo-

nententyp zugeordnete Typdomäne, da sich die Wissensbasis in Form des Assoziationsgeflechts (Topic Map) lediglich auf diese bezieht.

Die Suche nach Basistypen wie auch nach Typarten basiert dabei auf dem semantischen Typbegriff, sodass im Suchergebnis alle Komponententypen der betroffenen Domäne aufgeführt werden, die ein entsprechendes Suchkriterium auf semantischer Ebene erfüllen. Als Suchkriterien sind die in Abschnitt 4.4.3.1 definierten Typausprägungen zugelassen. Im Anschluss daran wird für jede, dem vorgegebenen Kriterium entsprechende Typausprägung deren syntaktische Realisierung auf der Nachrichtenebene wie auch auf der syntaktischen Ebene analysiert, indem gegen den ausgewählten Komponententyp abgeglichen wird. Letztendlich kann ein `Report`-Objekt zurückgegeben werden, das das Ergebnis der Typsuche beinhaltet.

Es bleibt noch anzumerken, dass für die Ableitung einer detaillierteren Aussage auf den Typvergleich zurückgegriffen werden muss. Das Application Programming Interface des Typmanagers stellt eine Methode zur Suche nach Basistypen zur Verfügung.

```
   public Report getBasetypes(ComponentType ct, boolean equals,
   boolean basetypes1,
boolean basetypes2, boolean basetypes2of1);

Parameter:
   ct - ein Komponententyp
   equals - suche gleiche Typausprägungen mit »true«
   basetypes1 - suche Typausprägungen des Basistyp-1 mit »true«
   basetypes2 - suche Typausprägungen des Basistyp-2 mit »true«
   basetypes2of1 - suche Typausprägungen des Basistyp-2von-1 mit »true«
gibt zurück:
   textbasierter Bericht der Typsuche
```

In Analogie zur Suche nach Basistypen kann ebenso nach Typarten gesucht werden.

```
   public Report getTypekinds(ComponentType ct, boolean equals,
   boolean typekinds1,
boolean typekinds2, boolean typekinds2of1);

Parameter:
   ct - ein Komponententyp
   equals - suche gleiche Typausprägungen mit »true«
   typekinds1 - suche Typausprägungen der Typart-1 mit »true«
   typekinds2 - suche Typausprägungen der Typart-2 mit »true«
   typekinds2of1 - suche Typausprägungen der Typart-2von-1 mit »true«
gibt zurück:
   textbasierter Bericht der Typsuche
```

Analog kann jetzt auch die Suche nach kompatiblen Komponententypen realisiert werden.

6.2.4.2 Kompatible Typen

Neben der Suche nach Basistypen sowie Typarten kann ebenso nach kompatiblen Komponententypen – also potenziellen Interaktionspartnern – eines ausgewählten Komponententyps gesucht werden. Dabei ist zu berücksichtigen, dass für den Fall einer polymorphen, typkonvertierbaren Komponente zunächst der relevante Anwendungskontext in Form einer festzulegenden Rolle spezifiziert werden muss, auf deren Grundlage nachfolgend die Suche nach kompatiblen Komponententypen vorgenommen werden kann.

Zur Auffindung kompatibler Komponententypen wird abermals der semantische Typ der Komponente in der relevanten Typdomäne zu Grunde gelegt. Für den Fall semantischer Kompatibilität kann wiederum für jeden im Suchergebnis aufgeführten Komponententyp dessen Nachrichtenebene wie auch dessen syntaktische Ebene im Hinblick auf Kompatibilität zum selektierten Komponententyp analysiert werden. Letztendlich wird das Suchergebnis abermals durch ein `Report`-Objekt geliefert.

Für die Ableitung einer detaillierteren Aussage muss auf die Kompatibilitätsprüfung verwiesen werden. Die folgende, über das Typmanager-API bereitgestellte Methode ermöglicht schließlich die Suche nach kompatiblen Komponententypen.

```
   public Report getCompatibleTypes(ComponentType ct);

Parameter:
   ct - ein Komponententyp
gibt zurück:
   textbasierter Bericht der Typsuche
```

Bemerkenswert ist im Folgenden eine Funktion zur Adaptersuche, mit der sich Adapterkomponenten finden lassen, die zwischen zwei zunächst inkompatible Komponenten geschaltet werden können, um deren typsichere Interaktion sicherzustellen.

6.2.5 Adapter

Adapter dienen der Überbrückung von Inkompatibilitäten zwischen Komponenten, die miteinander gekoppelt werden sollen. Sie werden aufgrund dieses Sonderstatus durch eine spezielle Domäne – die Adapterdomäne – von anderen Komponententypen abgegrenzt. Somit muss im ersten Schritt neben der Auswahl zweier zu koppelnder, inkompatibler Komponenten die der Adaptersuche zu Grunde liegende Adapterdomäne festgelegt werden. Hierfür muss abermals – analog zur Suche kompatibler Kompo-

nententypen – für typkonvertierbare Komponenten ein konkreter Anwendungskontext, in dem die Komponentenkopplung vorgenommen werden soll, spezifiziert werden. Dies geschieht wiederum durch die Angabe einer Rolle.

Ein Algorithmus, der die Adaptersuche implementiert, kann somit in der ausgewählten Domäne nach entsprechenden Adaptern suchen, wobei für jede Adapterkomponente die *beidseitig-vollständige Kompatibilität* hinsichtlich der beiden selektierten Komponenten sichergestellt werden muss. Ein entsprechender Algorithmus, der das Suchergebnis wiederum in Form eines `Report` zurückgibt, wird im Folgenden vorgestellt.

6.2.5.1 Prüfung beidseitig-vollständiger Kompatibilität

Der vorliegende Algorithmus zur Prüfung beidseitig-vollständiger Kompatibilität für Adapterkomponenten zielt auf die Beantwortung der Frage ab, ob ein Adapter zwischen zwei (im Regelfall syntaktisch) inkompatible Komponenten geschaltet werden kann, um deren typsichere Interaktion sicherzustellen. Dabei ist noch zu erwähnen, dass der Algorithmus prinzipiell nicht auf Adapterkomponenten mit genau zwei (abstrakten) Schnittstellen auf semantischer Ebene beschränkt ist. Dennoch sollte er im Kontext einer beidseitig-vollständigen Kompatibilitätsprüfung lediglich in Verbindung mit Adapterkomponenten verwendet werden, die eben genau zwei (abstrakte) Schnittstellen auf semantischer Ebene besitzen, da durch diese Forderung für die beiden über den Adapter gekoppelten Komponenten stets deren ordnungsgemäße Interaktion gewährleistet werden kann (s. Abschnitt 4.7).

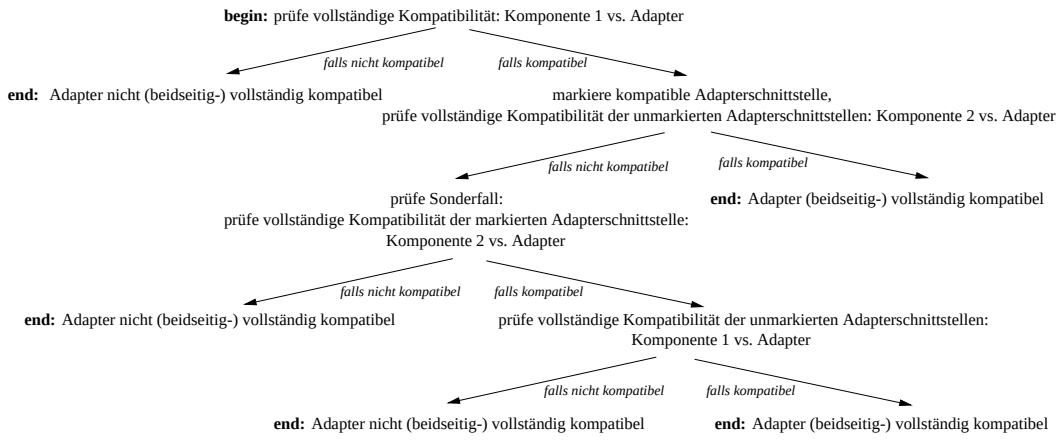
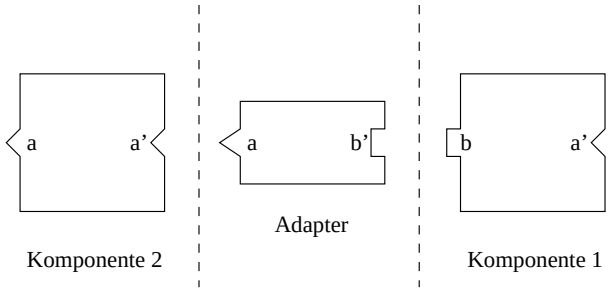


Abbildung 6.9
Ablauf des Prüfalgorithmus

Der Prüfalgorithmus analysiert zunächst die vollständige Kompatibilität der ersten Komponente gegen die Adapterschnittstellen. Tritt der Fall der vollständigen Kompatibilität ein, wird die zu Grunde liegende Adapterschnittstelle markiert und die zweite Komponente im Hinblick auf die unmarkierten Adapterschnittstellen auf vollständige Kompatibilität geprüft. Im Erfolgsfall kann der Adapter (beidseitig-) vollständig kompatibel zwischen beide Komponenten geschaltet werden. Sollte diese Prüfung nicht erfolgreich sein, muss der Sonderfall berücksichtigt werden, dass eventuell zufällig eine falsche erste Schnittstellenzuordnung vorgenommen wurde.

Die markierte Adapterschnittstelle wird daraufhin auf vollständige Kompatibilität bezüglich der zweiten Komponente sowie die unmarkierten Adapterschnittstellen hinsichtlich der ersten Komponente geprüft. Sollte in beiden Fällen die vollständige Kompatibilität gegeben sein, so kann der Adapter (beidseitig-) vollständig kompatibel zwischen beide Komponenten geschaltet werden.

Abbildung 6.10
(beidseitig-) vollständig kompatibler Adapter



In Abbildung 6.10 kann die Adapterkomponente (beidseitig-) vollständig kompatibel zwischen die beiden Komponenten geschaltet werden, indem die Adapterschnittstelle *a* mit der Komponentenschnittstelle *a'* der zweiten Komponente sowie die Adapterschnittstelle *b'* mit der Komponentenschnittstelle *b* der ersten Komponente gekoppelt wird.

Falls der Algorithmus bei der beidseitigen Kompatibilitätsprüfung (zufällig) zunächst die Schnittstelle *a'* der ersten Komponente der Adapterschnittstelle *a* zuordnet, wird vorerst eine falsche Schnittstellenzuordnung vorgenommen, denn die zweite Komponente ist bezüglich der Adapterschnittstelle *b'* nicht kompatibel. Dieser Sonderfall wird ebenfalls vom Prüfalgorithmus erkannt und behandelt.

Die entsprechende Funktion zur Adaptersuche wird wiederum über das API des Typmanagers bereitgestellt. Allerdings muss vor deren Aufruf zunächst die der Suche zu Grunde liegende Adapterdomäne festgelegt werden.

```
   public Report findAdapters(ComponentType ct1, ComponentType ct2);

Parameter:
   ct1 - ein Komponententyp
   ct2 - ein Komponententyp
gibt zurück:
   textbasierter Bericht der Adaptersuche
```

Alternativ können die Funktionen zur Typsuche wie auch die Adaptersuche ebenso mittels der grafischen Benutzeroberfläche des *Type Manager Explorer* über den Menüpunkt »Search« aufgerufen werden. Der *Basetype Report* sowie der *Typekind Report* analysieren dabei die zu dem ausgewählten Komponententyp (semantisch) korrelierenden Komponententypen im Hinblick auf Basistyp- beziehungsweise Typart-Beziehungen mit deren syntaktischer Realisierung. Der *Compatible Types Report* umfasst dagegen alle (semantisch) kompatiblen Komponententypen eines selektierten Komponententyps, wobei auch hier die entsprechenden syntaktischen Realisierungen analysiert werden. Dabei werden – analog zu den Typoperationen – Farbinformationen zur Visualisierung eingesetzt. Zur Durchführung einer Adaptersuche muss zunächst über die *Adapter Search Properties* eine Adapterdomäne spezifiziert werden, auf deren Basis eine nachfolgende Suche durchgeführt werden kann, wobei das resultierende Suchergebnis im *Adapter Search Report* ausgegeben wird. Über den Button »Store« kann der Bericht jeweils wieder als Textdatei gespeichert werden, während »Reject« den Bericht verwirft.

Zusätzlich zu dieser Basisfunktionalität der Typmanagement-Komponente sollen im Folgenden noch einige Zusatzfunktionen vorgestellt werden, die den Funktionsumfang der Typmanager-Applikation sinnvoll erweitern.

6.2.6 Zusatzfunktionen

Im Hinblick auf eine Erweiterung der Funktionalität des Typmanagers werden an dieser Stelle die Verwaltung von Typdomänen sowie von benutzerdefinierten Assoziationen und die Visualisierung der Typbeschreibung diskutiert. Zum implementierten Komponentenpool (Repository) siehe auch Abschnitt 6.3.3.

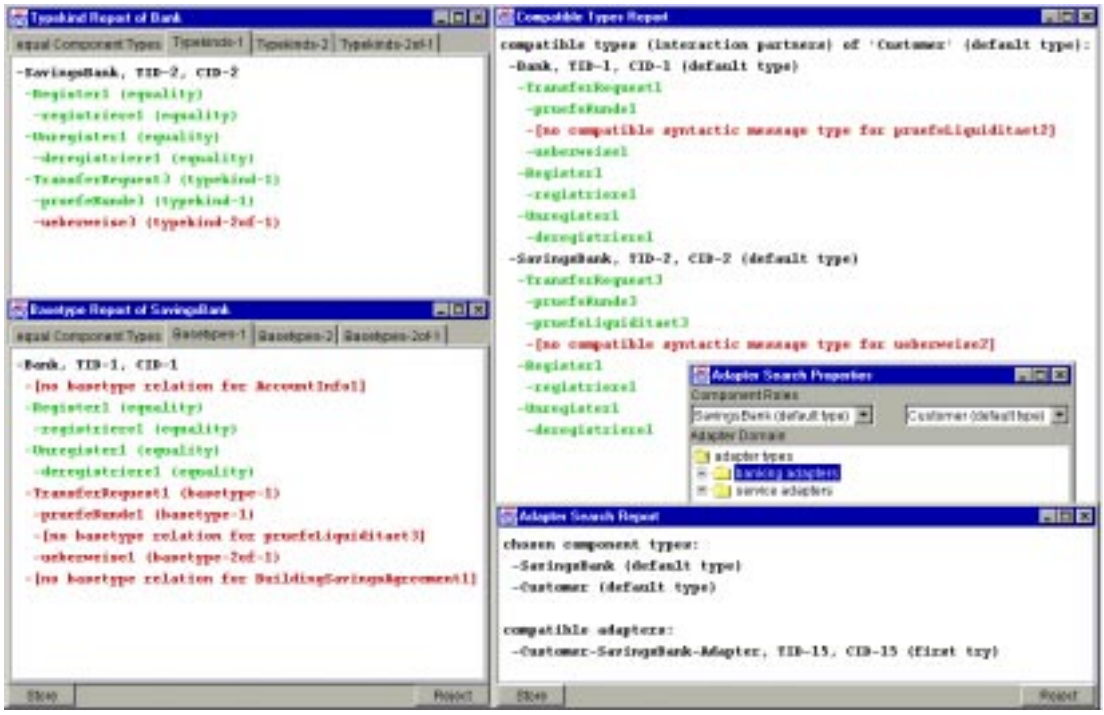


Abbildung 6.11
Suchfunktionen des Type Manager Explorer

6.2.6.1 Typdomänen

Typdomänen erlauben die Zusammenfassung von Komponententypen mit dem Hintergrund der Abgrenzung von Komponententypen anderer Domänen, wobei Typvergleiche sowie Kompatibilitätsprüfungen ausschließlich innerhalb einer Domäne als sinnvoll erscheinen und entsprechend nur dort durchgeführt werden. Das API der Typmanagement-Komponente stellt dabei Methoden zu deren Verwaltung (Anlegen, Löschen, ...) bereit. Hier soll exemplarisch die API-Methode zum Anlegen von Typdomänen vorgestellt werden.

```
   public void defineTypeDomain(String name, String path,
   boolean adapter);

Parameter:
   name - Name der Typdomäne
   path - Domänenpfad
   adapter - Kennzeichnung als Adapterdomäne mit »true«
```

Der *Type Manager Explorer* erlaubt dabei in seinem Hauptfenster (s. Abbildung 6.7) eine Navigation durch hierarchisch aufgebaute Namensräume in Form von Typdomänen, deren Verwaltung über den Menüpunkt »Administration« ermöglicht wird.

6.2.6.2 Benutzerdefinierte Assoziationen

In der vorliegenden Konzeption haben benutzerdefinierte Assoziationen die Bedeutung, die Notwendigkeit einer Adaption für die Elemente zweier Typbäume explizit auszudrücken. Hieraus ergibt sich die Anforderung, eine Liste noch zu entwickelnder Adapter zu verwalten. Dabei sind benutzerdefinierte Assoziationen auf jeder der drei Ebenen des Typbaums ausschließlich für die Wurzel-Elemente zulässig.

Die Adressierung der zu adaptierenden Elemente wird dabei durch den Pfad im Typbaum in der Form `ComponentType:SyntBEType:SyntMessageType` realisiert, wobei stets nur Elemente der gleichen Ebene in Beziehung gesetzt werden dürfen. Das API des Typmanagers stellt Methoden zur Verwaltung benutzerdefinierter Relationen bereit. An dieser Stelle wird exemplarisch die Etablierung derartiger Assoziationen aufgezeigt.

```
   public void defineUserRelation(String path1, String path2);

Parameter:
   path1 - Pfad zu einem Wurzel-Element des Typbaums
   path2 - Pfad zu einem Wurzel-Element des Typbaums
```

Der *Type Manager Explorer* stellt die entsprechende Funktionalität über den Menüpunkt »Extras« bereit, wobei benutzerdefinierte Assoziationen über »Add User Defined Relation« hinzugefügt beziehungsweise ebenso aus der Tabelle aller definierten Relationen »User Defined Relations« entfernt werden können.

6.2.6.3 Ansicht der Typbeschreibung

Zur Visualisierung der internen Struktur des Komponententyps (Typbaum) werden nachfolgend zwei Wege zu deren Realisierung vorgestellt. Zum einen steht die relativ primitive Funktion der Ausgabe der Typbeschreibung auf Kommandozeilenebene zur Verfügung, für die eine entsprechende API-Methode existiert.

```
   public void showTypehierarchy(ComponentType ct);

Parameter:
   ct - ein Komponententyp
```

Zum anderen kann eine Visualisierung der Typbeschreibung im *Type Manager Explorer* vorgenommen werden, indem der relevante Komponententyp selektiert und im Anschluss daran über den Menüpunkt »Extras« der Untermenüpunkt »Show Type Description« aufgerufen wird. Die grafische Benutzeroberfläche erlaubt insbesondere die Navigation durch die Baumstruktur der

hierarchisch aufgebauten Typbeschreibung, wobei terminale Elemente (Blätter) des Typbaums die Ansicht ihrer internen Repräsentation gestatten. Terminale Elemente sind in diesem Kontext Protokolle in Form endlicher Automaten, Merkmalsstrukturen, Nachrichten- sowie Operationstypen.

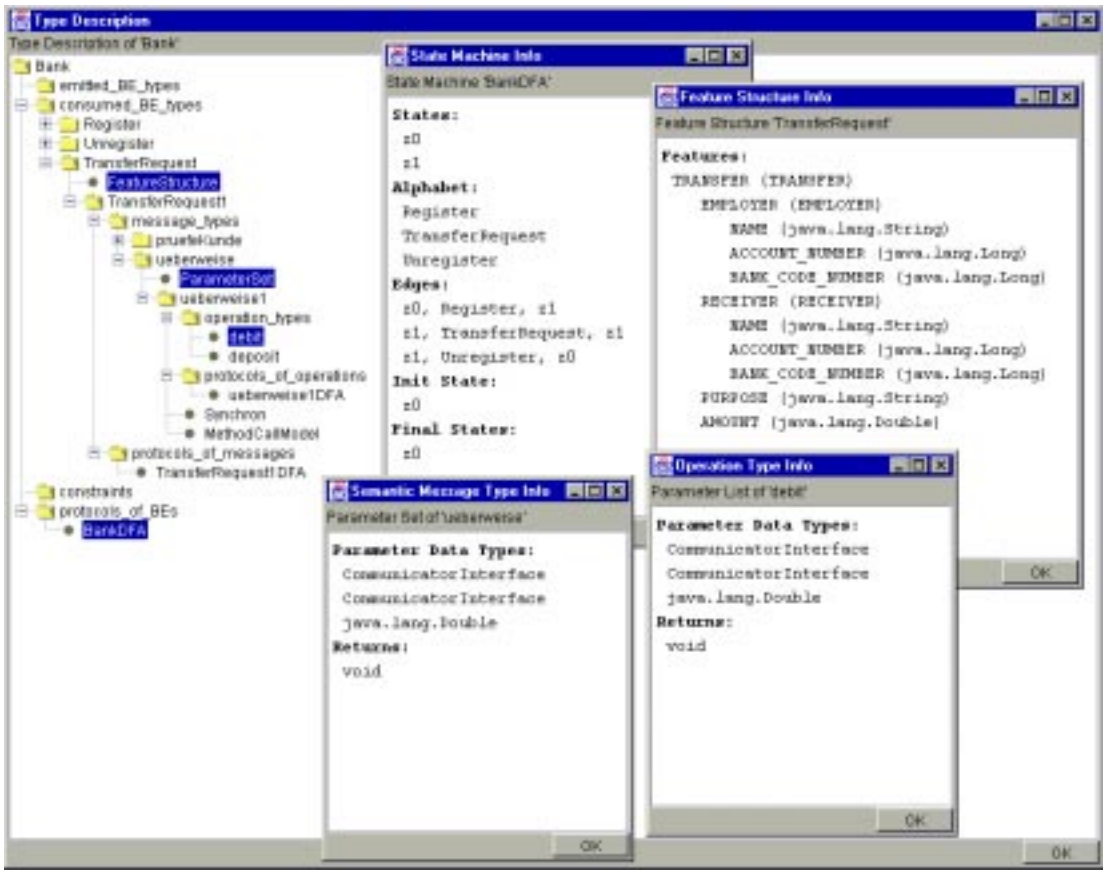


Abbildung 6.12
Ansicht der Typbeschreibung im Type Manager Explorer

Nicht implementiert wurde an dieser Stelle eine Funktion zur Rückgabe der XML-Repräsentation der Typbeschreibung[12]. Diese könnte allerdings nachträglich auf Basis einer Methode `getTypeDescription(...)` realisiert werden, die sich dabei auf die durch den XML-Parser aufgebauten Document Object Models (DOMs) stützt. Voraussetzung dafür wäre allerdings die Persistenz der DOMs (derzeit ist nur die interne Repräsentation der Wissenbasis persistent).

[12] Es wurde davon ausgegangen, dass solche Beschreibungen ja bereits existieren, da gerade sie dem Typmanager als Eingabe dienen.

Eine Darstellung des Application Programming Interface der Typmanagement-Komponente fasst nachfolgend noch einmal alle bereits vorgestellten API-Methoden zusammen, wobei die Schnittstelle weitere Methoden zur Verfügung stellt, auf die im Einzelnen bisher nicht eingegangen wurde.

6.2.7 API des Typmanagers

Die folgende Abbildung zeigt das Application Programming Interface (API) der Typmanagement-Komponente. Das Interface stellt dabei Operationen zum Registrieren und Löschen von Komponententypen zur Verfügung und erlaubt Typvergleiche sowie Kompatibilitätsprüfungen. Weiterhin ist die Schnittstelle um diverse Suchfunktionen ergänzt, um Typarten, Basistypen sowie kompatible Komponententypen auf der Grundlage eines ausgewählten Komponententyps zu finden. Ferner bietet das Interface die Mäglichkeit der Adaptersuche.

Zur strukturierten Ablage von Komponententypen können Typdomänen angelegt werden, die über die Schnittstelle verwaltet werden können. Dabei ergibt sich die Möglichkeit der Navigation durch hierarchisch strukturierte Namensräume. Die interne Struktur des Komponententyps (Typbaum) ist schließlich vom Typmanager-Benutzer einsehbar, wobei dieser zusätzlich benutzerdefinierte Assoziationen zwischen Typrepräsentationen festlegen sowie diese verwalten kann. Zudem kann über das Interface auf die Wissensbasis und die Komponententabelle (Pool) zugegriffen werden. Die Sicherung der Wissensbasis des Typmanagers garantiert letztendlich eine persistente Datenhaltung.

***Listing 6.31**: Typmanager-API*

```
public interface TypeManagerInterface
{
  // Registrieren und Löschen von Typbeschreibungen
  public void register(String filename, boolean adapter, String domainpath);
  public void removeComponentType(ComponentType ct, boolean adapter);

  // Typvergleiche und Kompatibilitätsanfragen
  public Report compare(ComponentType ct1, ComponentType ct2);
  public Report compareSyntBETypes(SyntBEType sbet1, SyntBEType sbet2);
  public Report checkCompatibility(ComponentType ct1, ComponentType ct2);

  // Suchfunktionalität
  public Report getTypekinds(ComponentType ct, boolean equals,
       boolean typekinds1, boolean typekinds2, boolean typekinds2of1);
  public Report getBasetypes(ComponentType ct, boolean equals,
       boolean basetypes1, boolean basetypes2, boolean basetypes2of1);
  public Report getCompatibleTypes(ComponentType ct);
  public Report findAdapters(ComponentType ct1, ComponentType ct2);
```

```
  // Typdomänen anlegen, löschen und verwalten
  public void defineTypeDomain(String name, String path, boolean adapter);
  public void removeTypeDomain(String name, String path, boolean adapter);
  public void setCurrentComponentDomain(Domain domain);
  public Domain getCurrentComponentDomain();
  public void setCurrentAdapterDomain(Domain domain);
  public Domain getCurrentAdapterDomain();

  // Ausgabe der Typbeschreibung
  public void showTypehierarchy(ComponentType ct);

  // benutzerdefinierte Beziehung setzen und löschen
  public void defineUserRelation(String path1, String path2);
  public void removeUserRelation(String path1, String path2)
               throws RelationNotFoundException;
  // Zugriff auf die benutzerdefinierten Beziehungen
  public Vector getUserRelations();

  // Zugriff auf die Wissensbasis und die Komponententabelle
  public XMLtoTypehierarchy getDatabase();
  public Hashtable getComponentTable();

  // Wissensbasis sichern
  public void saveTypeManager();
}
```

6.2.8 Modularisierung der Implementation

Durch die Identifikation der obigen unterschiedlichen Funktionsbereiche der Typmanagement-Komponente ist die Modularisierung der Implementation bestimmt. Der modulare Aufbau erleichtert dabei zukünftige Modifikationen der Implementation, da die verschiedenen Module ihre Implementationsdetails nach aussen verbergen (Kapselung), sodass sich potenzielle Änderungen lediglich auf das betroffene Modul auswirken. Damit folgt die Implementation selbst den in Kapitel 2 zu Grunde gelegten Prinzipien eines segmentierten Komponentenmodells. Folgende Segmente ergeben sich:

- `TypeManager` (Basismodul: implementiert das `TypeManagerInterface`)
  - Typregistrierung
  - Löschen von Komponententypen
  - Sicherung der Wissensbasis
  - Koordination von API-Aufrufen: Vermittlung an die zuständigen Module
- `TypeOperationManager`
  - Typvergleich

  - Kompatibilitätsprüfung
- `SearchManager`
  - Suche nach Typarten, Basistypen
  - Suche nach kompatiblen Typen
  - Adaptersuche
- `DomainManager`
  - Verwaltung der Typdomänen
- `ConsoleTypeDescriptionViewer`
  - Ausgabe der Typbeschreibung auf Kommandozeilenebene
- `UserRelationManager`
  - Verwaltung benutzerdefinierter Beziehungen
- `XMLtoTypehierarchy`
  - Parsen der XML-Typbeschreibung: Abbildung auf den (internen) Typbaum

Im Abbildung 6.13 ist die entspechende Übertragung auf ein UML-Klassendiagramm für die implementierten Typmanagement-Komponente dargestellt.

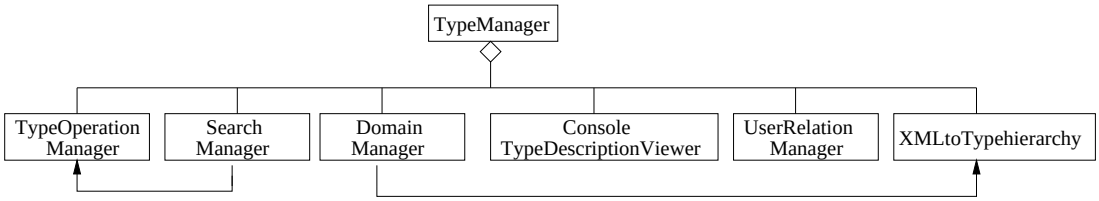


Abbildung 6.13
Segmentierter Aufbau des Typmanagers

Das Klassendiagramm zeigt die Klasse `TypeManager`, die die zu den einzelnen Segmenten korrespondierenden Klassen – die Funktionsbereiche des Typmanagers – aggregiert. Die Nutzungsbeziehungen zwischen den Klassen `TypeOperationManager` und `SearchManager` sowie den Klassen `DomainManager` und `XMLtoTypehierarchy` sind erkennbar.

Mit der vorgestellten Implementation des Typmanagement steht nun die wichtigste Grundlage zur Realisierung eines generativen Prozesses bereit, dessen Umsetzung der nachfolgend beschriebenen Implementation des Generators obliegt.

6.3 Der Generator

Die Umsetzung und Arbeitsweise der in Abschnitt 5.8 dargestellten Mechanismen und Konzepte des Meta-Generators werden im Folgenden detaillierter vorgestellt, wobei zunächst ein Überblick über die Implementation des Prototypen gegeben wird. Im Anschluss wird auf die einzelnen Komponenten des Generators selbst eingegangen. Es soll weniger auf den konkreten Code als mehr auf die Zusammenhänge der implementierten Funktionen eingegangen werden. Dabei ist die Arbeitsweise des Generators auf drei Aufgabengebiete verteilt, denen jeweils einzelne Bausteine des Generators zugeordnet werden können. Im ersten Schritt erfolgt die Charakterisierung der gewünschten Anwendung. Diese Phase betrifft die Bestandteile `Generator`, `ComponentPool` und `FeatureGUI`.[13] In Abschnitt 6.3.4 wird das Zusammenspiel dieser Komponenten erörtert. Die zweite Phase betrifft die Erstellung der konkreten Anwendungsstruktur. Hierfür muss aus den ausgewählten Komponenten und der DTD, die als Architekturstil alle möglichen Anwendungsstrukturen einer Produktlinie aus einer Domäne festlegt, eine konkrete Struktur generiert werden. Diese Aufgabe hat der `XMLDocBuilder`. Die letzte Phase umfasst die Anwendungserstellung unter zu Hilfe nahme der konkreten, dann ebenfalls als XML-Dokument vorliegenden Anwendungsstruktur und den ausgewählten Komponenten, sowie das zugehörige Customizing im Rahmen der Post-Konfiguration. Neben diesen Generierungsphasen werden auch die abschließenden Erweiterungen der eingesetzten XML-Komponentendeskriptoren vorgestellt.

Die für die Illustration einer konkreten Anwendungserstellung notwendigen Komponenten könnten auf Grund der Gesamtkonzeption des Ansatzes durch unterschiedlichste Techniken realisiert sein. Um dies auch in der Praxis nachzuweisen, sind im Rahmen dieser Arbeit in der Tat Beispielkomponenten mit unterschiedlichen – insbesondere auch verteilten – Techniken aus dem Umfeld der Firmen Sun und Microsoft sowie der OMG realisiert worden (vgl. [Benthien, 1999], [Martens, 1999], [Moehl, 1999b], [Lapok, 2000], [Häming, 1999], [Bernhardt & Djokić, 2001] und [Hühnken, 2001]), deren vollständige Darstellung hier den Rahmen sprengen würde, sodass in diesem Abschnitt im Wesentlichen die am kompaktesten darstellbaren *JavaBeans* zur Illustration verwendet werden. Das unterschiedliche Komponententechnologien und von ihnen genutzte Kommunikationsparadigmen

[13] Komponenten- und Klassennamen werden wie schon zuvor in `Typewriter-Schrift` wiedergegeben.

vom Generator insbesondere auch *gemeinsam* – und damit mit einem hohen Wiederverwendungsgrad – zur Applikationserstellung genutzt werden können, wurde im Rahmen dieser Arbeit in [Bernhardt & Djokić, 2001] gezeigt.

6.3.1 Technische Kurzbeschreibung

Die Hauptkomponente des Generators wird sinnvollerweise mit `Generator` bezeichnet. Diese leitet den eigentlichen Generierungsvorgang ein und steuert die notwendigen Arbeitsschritte. Sie vermittelt zwischen den verschiedenen Bestandteilen des Gesamtsystems. In ihr ist damit die eigentliche »Intelligenz« des Generierungsvorgangs angesiedelt. Den notwendigen grafischen Rahmen mit Menuführung und der Möglichkeit der grafischen Ausgabe bietet die Komponente `MenuWindow`. Beim Start der Generierung wird intern der `ComponentPool` gefüllt, der als Generator-spezifischer Cache (Zwischenspeicher zur Zugriffsoptimierung) Teil des eigentlichen Komponentenrepository ist. Der `ComponentPool` kapselt aus Sicht des Generators insbesondere die Typmanagmentschnittstellen zur Auswahl bzw. Abfrage der Merkmale vorhandener Komponenten.[14]

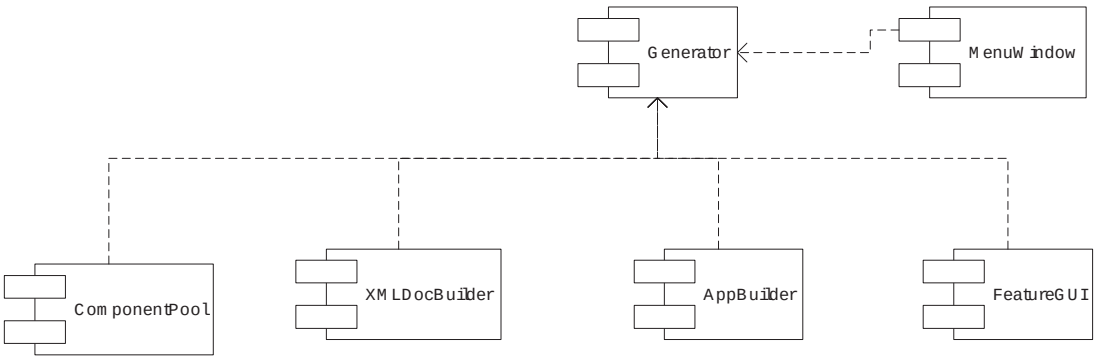


Abbildung 6.14
Komponenten des Generators

Vor dem Start der Generierung legt der Softwarekonstrukteur fest, welche Art von Anwendung er generieren möchte.

[14] Diese Entkopplung erlaubt insbesondere auch den vollständigen Austausch des genutzten Typmanagements. So stand für den ursprünglichen im Rahmen dieser Arbeit in [Kröhling, 2000] entwickelten Prototypen zum Beispiel noch keine Typmanager-Implementation zur Verfügung. Diese konnte jedoch nach ihrer Erstellung auf Grund der Komponentenstruktur des Generatorsdesigns innerhalb kürzester Zeit im Rahmen von [Bernhardt & Djokić, 2001] integriert werden.

Für die angeleitete Benutzerführung des anschließenden Konkretisierung der Applikation wird eine Priorisierung einzelner Komponententypen für die gewählte Domäne benötigt (vgl. Abschnitt 5.10.4). Die Komponente `Sequence` liest hierfür ein vom Merkmalseditor (s. Abschnitt 6.4.3) erstelltes XML-Dokument ein und bietet dem Generator eine Schnittstelle für die Abfrage der entsprechenden Prioritäten an.

Die Auswahl der gewünschten Merkmale eines Komponententyps wird durch die Komponente `FeatureGUI` vorgenommen. Der `Generator` baut dabei für jeden in Frage kommenden Komponententyp eine Oberfläche mit Karteikarten auf. Jedes Merkmal wird als eine anklickbare Schaltfläche (engl. *Button*) dargestellt. Wenn ein Merkmalswert ausgewählt wurde, wird der `Generator` darüber informiert und fragt über den `ComponentPool` nach allen Merkmalen, die noch zur freien Auswahl stehen, wenn die bisher ausgewählten Merkmalswerte berücksichtigt werden (vgl. Abschnitt 6.3.4). Ein Merkmalswert steht noch zur freien Auswahl, wenn noch kein anderer Wert unter der jeweiligen Merkmalsbezeichnung ausgewählt wurde und eine Komponente (im Repository) existiert, die alle bis dahin ausgewählten Merkmale, alle für den zugehörigen Komponententyp signifikanten Merkmale (vgl. Abschnitt 5.10.2) und eben diesen gewünschten Merkmalswert anbietet. Alle nicht zur Verfügung stehenden Werte werden deaktiviert dargestellt, sodass keine unmöglichen Merkmalskombinationen ausgewählt können.

Wenn der Konstrukteur alle für ihn wichtigen Werte[15] angegeben hat, kann er die Auswahl bzw. Suche nach einer konkreten Komponente des jeweiligen Komponententypes starten, die die festgelegten Anforderungen erfüllt. Der Generator gibt diese Anfrage über den `ComponentPool` an das vom Typmanagement verwaltete Repository weiter und findet auf jeden Fall mindestens eine Komponente, die die Spezifikation erfüllt.[16] Wenn mehrere Komponenten gefunden werden, wählt der `ComponentPool` in der hier beschriebenen Prototypimplementation eine da-

[15] Diese sind nicht im Sinne von variabel belegbaren Werten gemeint (das entspräche der Einstellung von Eigenschaften – Properties – einer Komponente), sondern kennzeichnen die fixen Einstellungen eines Merkmals.

[16] Sonst hätte die zugehörige Mermalsauswahl gar nicht stattgefunden, da die angebotenen Merkmale aus den zur Verfügung stehenden Komponententypen abgeleitet wurden. Die Merkmale der in Abschnitt 5.12.6 eingeführten abstrakten Typen bleiben dabei unberücksichtigt, sodass auf jeden Fall immer eine konkrete Komponente registriert wurde, wenn eine entsprechende Typbeschreibung vorliegt.

von zufällig aus. Es gibt derzeit nicht die Möglichkeit zusätzliche Kriterien für diese Auswahl der Komponente anzgeben. Es wäre aber denkbar, allgemeine Kriterien wie die Arbeitsgeschwindigkeit oder den Preis einer Komponente einfließen zu lassen und etwa im Sinne eines Trading [Griffel, 1998] zu behandeln.[17]

Der Generator hat zu diesem Zeitpunkt der Verarbeitung keine Kenntnis darüber, ob die Auswahl einer Komponente die Wahl später auszuwählender Komponententypen durch die bislang übernommenen Merkmale verhindert, da einzelne Merkmale die Auswahl mehrerer Komponenten beeinflussen können, d.h. signifikante Features für die Auswahl anderer Komponententypen sind (vgl. Abschnitt 5.10.2). Es ist auch keine Entscheidungsrücknahme im Sinne etwa eines »backtracking« vorgesehen, da die Architekturhierarchie und eventuell durch eine Domäne vorgegebenen Priorisierungen von Komponenten[18] akzeptiert und durch den Konstrukteur nicht in Frage gestellt werden sollen. Der Spezifikationsvorgang der Merkmale wiederholt sich für jeden für die Anwendungsstruktur in Frage kommenden Komponententyp.

Sobald alle Komponententypen durch entsprechende Merkmalsauswahl konkretisiert wurden, startet der `XMLDocBuilder` (s. Abschnitt 6.3.5), der die Aufgabe hat, aus einer Liste mit den ausgewählten Komponenten und einer DTD, die die Struktur möglicher Anwendungen der Domäne beschreibt, ein für eben diese DTD gültiges XML-Dokument zu erstellen (»assemblydescriptor«, vgl. a. Abschnitt 6.22). Jede ausgewählte Komponente entspricht hierbei einem Element, das mit dem Namen des jeweiligen Komponententypen gekennzeichnet wird.

Das erstellte XML-Dokument bildet dann die Grundlage für den nächsten Arbeitsschritt. Der `AppBuilder` (s. Abschnitt 6.3.6.2) liest dieses XML-Dokument ein und konstruiert aus den enthaltenen Informationen, sowie den Schnittstellenbeschreibungen der Komponenten aus deren Typangaben, den Quellcode eines Anwendungsgerüstes.[19] Der `AppBuilder` ge-

[17] Nach Abschluss der vorliegend dokumentierten Arbeiten ist dieser Ansatz im Rahmen eines industriellen Kooperationsprojekts zur Erstellung einer Geldautomaten-Software in der Tat erprobt worden.

[18] D.h. der von einem Domänenspezialisten festgelegte Grad der Bedeutsamkeit einer Komponente für eine bestimmte Anwendungsart wird berücksichtigt.

[19] Der Generatorprototyp ist derzeit ausschließlich in der Lage Java-Code zu erzeugen. Die Einbindung sprachfremder Komponenten erfolgt über entsprechende *Bridges* (siehe etwa `http://developer.java.sun.com`), die sich homogen in das ohnehin vorhandene Adapterkonzept

neriert zugleich Quellcode, der alle ausgewählten Komponenten instanziiert[20], für eine eventuell benötigte visuelle Darstellung sorgt und die Komponenten den geltenden Randbedingungen entsprechend, d.h. unter Aktivierung des entsprechenden Policymanagements (vgl. Abschnitt 3.10) miteinander verknüpft. Zudem hat der Konstrukteur noch die Möglichkeit, zusätzliche Eigenschaften der einzelnen Komponenten in der Post-Konfiguration zu setzen. Hierfür werden die individuellen, zu einer eingesetzten Komponente gehörigen Werkzeuge (Property-Editoren, Customizer, vgl. [Griffel, 1998]) für die Komponenten angezeigt. Diese erlauben es, solche bereits vom Komponentenentwickler vorgesehenen Variationspunkte ebenfalls konsistent zu bearbeiten. Um eine lauffähige Anwendung zu erhalten, ist es daher auch ausreichend, die voreingestellten Standardwerte der entsprechenden Eigenschaften zu verwenden.

Gemäss der Hierarchieforderung (s.a. Abschnitt 5.9.2) werden nur Komponenten miteinander verknüpft, wenn sie im Baum des XML-Dokumentes benachbart sind. Hierbei werden die Ein- und Ausgänge der beiden Komponenten miteinander verglichen und bei Kompatibilität miteinander verknüpft. Die eine Komponente muss hierfür Nachrichten liefern, für deren Empfang die andere implementiert ist. Die hier dokumentierte Implementation setzt als Zielmodell die JavaBeans und zur Nachrichtenrepräsentation die zugehörigen Java-Events ein. Einen Sonderfall stellt dabei dann das `ActionEvent` dar[21], da hierfür eine Adapterklasse generiert wird, die es einer Komponente ermöglicht, mit mehreren verschiedenen anderen Komponenten verknüpft zu werden und die `ActionEvents` zu unterscheiden, wie es etwa im Falle von Buttons für grafische Oberflächen notwendig ist. Eine der einzusetzenden Komponenten kann dann eine entsprechende GUI-Komponente sein. Eine solche GUI-Komponente bietet Methoden (in dieser Arbeit als *GUISlots* abstrahiert) für ver-

---

einfügen lassen. Die Austauschbarkeit der `AppBuilder`-Komponente würde aber auch andere Zielsprachen erlauben.

[20] Dies ist zumindest immer dann erforderlich, wenn die konkreten Komponenten nicht zum Beispiel im Sinne verteilter Dienste ohnehin bereits in einem Netzwerk aktiv sind (vgl. a. [Griffel, 1996]).

[21] Allgemeiner gesagt, besteht der Sonderfall in einem visuellen Layout zu unterwerfenden Komponenten, da ein solches Layout dem beschriebenen Generator (etwa im Gegensatz zu [Griffel, 1996]) nicht vorgegeben wird, sondern es ebenfalls aus den Deskriptorinformationen abgeleitet wird. Hier wird jedoch nur die spezielle Situation für JavaBean-Komponenten betrachtet.

schiedene Komponententypen an, um andere visuelle Komponenten anzeigen (»aggregieren«) zu können.

Wenn der Quellcode fertig generiert ist, muss dieser kompiliert werden. Die generierten Dateien sowie alle anderen für diese Anwendung benötigten Informationen werden in ein Anwendungsarchiv (im Falle der Java-Implementation ein ausführbares jar-File) verpackt. Es werden dabei auch die abhängigen Dateien berücksichtigt, die die einzelnen Komponenten in ihren Deskriptoren angeben, sodass eine konsistente, vollständige und direkt der Einsatzkonfiguration zuführbare Applikation entsteht.

6.3.2 Auswahl der Komponenten

Die Komponente `Generator` trägt die Hauptfunktionalität des Generierungsvorganges. Hier werden alle benötigten Hilfskomponenten instanziiert und die Phasen der Anwendungskonstruktion gelenkt. Der `Generator` vermittelt zwischen dem `ComponentPool`, der Abfragen über die wirklich vorhandenen Komponenten erlaubt, und der grafischen Oberfläche zur Merkmalsauswahl. Sie bestimmt, was als jeweils nächstes bei einem Genertordurchlauf zu tun ist. Für den Aufbau des XML-Dokumentes, das die Anwendungsstruktur repräsentiert, wird der `XMLDocBuilder` hinzugezogen. Das hieraus entstehende Dokument dient dem `AppBuilder` als Grundlage der Erstellung des Anwendungsgerüsts. Der `Generator` erkennt auf Basis der ihm vorgegebenen Produktlinie, wann alle zugehörigen Komponenten spezifiziert sind und wie die nachfolgenden Arbeitsschritte auszuführen sind. Der `Generator` startet zuallererst mit dem `MenuWindow` seine interaktive Oberfläche. Der `ComponentPool` und die `Sequence`, die die Reihenfolge der dem Benutzer zur Merkmalsauswahl vorgelegten Komponentypen lenkt, werden instanziiert.

6.3.2.1 Reihenfolge der Merkmalsspezifikation

Die Hilfskomponente `Sequence` bietet eine Schnittstelle, um den als nächsten zu konkretisierenden Komponententyp »abzufragen«, d.h. seine Merkmale auszuwählen. Die Informationen hierfür erhält die Hilfskomponente aus einem XML-Dokument, das jeweils für das mittels des Merkmalseditors erstellten Modells einer Produktlinie spezifisch ist. Das XML-Dokument kann als ein Baum angesehen werden, bei dem jedes Element einem Komponententyp entspricht, der abgefragt werden soll. Jedes Element enthält Informationen über den Namen des benötigten Kompo-

nententyps, sowie Informationen, wie häufig dieser ausgewählt werden soll. Es wird genau ein Pfad durch den Baum von der Wurzel zu einem Endblatt abgefragt (vgl. a. Abschnitt 5.5.2), was genau einer konkreten Ausprägung eines Produktes im Rahmen der architektonischen Variabilität entspricht. Auf diese Weise kann sich die Auswahlmöglichkeit von Komponententypen einer Produktlinie unterscheiden.

Wenn der XML-Baum an einer Stelle mehr als einen Sohn hat, ist eine Entweder-Oder-Entscheidung zwischen zwei oder mehr Komponententypen zu treffen. Bei Entweder-Oder-Entscheidungen wird dem Konstrukteur durch Informationen in Form textueller Beschreibungen der Konsequenzen der Entscheidung auf die Spezifikation der Anwendung geholfen. Es erscheint dann ein Auswahlfenster mit allen möglichen Komponententypen, die als nächstes zur Auswahl stehen. Zu jedem Komponententypen liegt eine textuelle Beschreibung der Konsequenzen der Auswahl vor. Diese Informationen stammen ebenso wie die möglichen Komponententypen aus dem eingelesenen XML-Dokument, das im Rahmen des Modellentwurfs erzeugt wurde (vgl. a. Abschnitt 5.3.4 und 6.4.3). Der Anwender kann genau einen Komponententyp auswählen, der dann als nächstes konkretisiert, d.h. in seinen Merkmalsausprägungen festgelegt werden kann. Die Wahl beeinflusst den weiteren Verlauf der angeleiteten Konfiguration, da der Zweig des Baumes weiterverfolgt wird, der den gewählten Komponententyp repräsentiert. Wenn eine solche Entscheidung getroffen wurde, muss die `Sequence` entsprechend darüber informiert werden.

Es sind nur Entweder-Oder-Entscheidungen, aber nicht Oder-Entscheidungen vorgesehen. Eine Oder-Entscheidung hätte die Form `(a|b)*` in der gewählten Architekturbeschreibung. Dieser Fall ist aber durch die Einschränkungen der Komponente `XMLDocBuilder` weder vorgesehen (s. Abschnitt 6.3.5.3, noch notwendig. Es wird stattdessen immer eine beliebige Anzahl beider Komponententypen nacheinander zugelassen (in DTD-Notation: `a*,b*`). Es stehen zu einem Zeitpunkt immer nur die Merkmale von Komponenten genau eines Komponententypes zur Auswahl.

6.3.2.2 Auswahl einer einzelnen Komponente

Der `ComponentPool` bietet eine Abfragemöglichkeit, welche Merkmale ein bestimmter Komponententyp umfasst. Es wird dabei herausgefiltert, welche Merkmale signifkant sind (vgl. Abschnitt 5.10.2) und welche zur freien Auswahl stehen. Für die in-

teraktive Merkmalsauswahl werden wie erwähnt nur die Merkmale berücksichtigt, die von Komponenten stammen, die alle bisher ausgewählten, signifikanten Merkmale implementieren. In einigen Fällen trifft der Generator eine automatische Entscheidung:

- Wenn nur genau eine Komponente gefunden wird, wird diese automatisch ausgewählt und der Konstrukteur hierüber informiert. Dies ist auch der Fall, wenn mehrere Komponenten des Typs von der Spezifikation her möglich gewesen wären. Der Generator geht danach zur Konkretisierung des nächsten Komponententyps über.

- Wenn keine Komponente gefunden wurde, die die signifikanten Merkmale für einen Komponententyp anbietet, und der Komponententyp optional ist oder beliebig oft vorkommen kann, d.h. auch kein mal, wird die Konkretisierung des Komponententyps automatisch übersprungen, da dann eine korrekte Anwendung erstellt werden kann, auch wenn keine Komponente des Typs[22] verwendet wird.

- Wenn keine Komponente gefunden wird, die die für diesen Komponententyp signifikanten Merkmale implementiert, eine Komponente dieses Types aber für die Anwendung notwendig ist, wird eine Fehlermeldung ausgegeben.

Die angeleitete Konfiguration verhindert zunächst die Definition von ungültigen Merkmalskombinationen für jeweils einen einzigen Komponententyp. Auswirkungen auf die mögliche Auswahl bei weiteren Komponententypen werden nur durch die Prioritätsvorgaben der Architekturspezifikation und die entsprechende Auswahlhistorie der Merkmale berücksichtigt. Trifft der Generator auf eine unmögliche Kombination, kann er nur mitteilen, welche Merkmale eine Komponente hätte erfüllen müssen. Er kann eine vorher getroffene Merkmalsauswahl derzeit nicht wieder rückgängig machen. Der Konstukteur muss beim nächsten Spezifikationsdurchlauf bei vorherigen Komponententypen eventuell andere Entscheidungen treffen. Dieser Fall tritt allerdings nur auf, wenn gar keine Komponente eines Typs im Komponentenpool vorhanden ist (etwa durch einen sytemtechnischen Ausfall des Repository) oder vorherige Entscheidungen die Auswahl verhindern.

[22] Wobei in diesem Fall eben auch keine passende Komponente zur Verfügung stünde.

Wenn mehrere Komponenten den bis dahin festgelegten Anforderungen entsprechen, wird die Konfigurationsoberfläche gemäß des Leitbildes aus Abschnitt 5.11.1 aufgebaut. Es wird die Methode `findComponent` beim Generator aufgerufen, die den Suchaufruf an den `ComponentPool` weitergibt. Das weitere Vorgehen hängt davon ab, ob der zu spezifizierende Komponententyp mehrfach vorkommen darf. Ist dies der Fall und es werden mehrere mögliche Komponenten gefunden, hat der Anwender die Möglichkeit, beliebig viele oder mindestens eine dieser Komponenten auszuwählen.

Es erscheint ein Fenster mit einer Kurzbeschreibung aller möglichen Komponenten, von denen eine oder mehrere ausgewählt werden können. Auf diese Weise kann der Anwender mehrere ähnliche Komponenten auf einmal auswählen. Wenn zum Beispiel ein Taschenrechner mit allen vorhandenen trigonometrischen Funktionseinheiten gewünscht wird, wird nur das Merkmal *Funktionsart=trigonometrisch* ausgewählt und es erscheint eine Auswahl aller trigonometrischen Funktionseinheiten. Die Auswahl erfolgt nicht automatisch, damit nicht mehrere Komponenten ausgewählt werden, die eigentlich die gleiche Aufgabe erledigen, aber ein, in diesem Fall vielleicht unbedeutenderes, unterschiedliches Merkmal anbieten. Bei Komponententypen, von denen nur eine einzige Komponente ausgewählt werden kann, wird automatisch eine der möglichen Komponenten ausgewählt. Es ist nicht sinnvoll, diese manuelle Auswahl auch bei diesen Komponententypen anzubieten, da dies den Automatisierungsgrad der Komponentenauswahl zu sehr reduzieren würde.

Wenn mehrere Komponenten eines Typs möglich sind, wird der Konstrukteur jeweils gefragt, ob weitere Komponenten dieses Types konkretisiert werden sollen. Ist dies der Fall, wird der Vorgang wiederholt. Andernfalls geht der `Generator` zur Bearbeitung des nächsten Komponententyps über. Der `Generator` führt eine Liste aller ausgewählten Komponenten und deren Merkmalen. Bei Komponententypen, die höchstens einmal vorkommen dürfen, vermerkt der `Generator` alle Merkmale der Komponente. In diesem Fall werden auch die Merkmale berücksichtigt, die nicht direkt vom Konstrukteur ausgewählt wurden, aber in der Typbeschreibung der ausgewählten Komponente spezifiziert sind.

Wenn alle Komponenten ausgewählt sind, startet der `Generator` den `XMLDocBuilder` zum Aufbau eines XML-Dokumentes des Anwendungsgerüstes für eben jene Komponenten unter Berücksichtigung der Document Type Definition für die gewählte Anwendungslinie. Danach startet der `Generator` den

`AppBuilder`, der aus der erstellten Anwendungsstruktur und den Schnittstellen der Komponenten eine Anwendung erstellt.

6.3.2.3 FeatureGUI

Die Komponente `FeatureGUI` bietet dem Konstrukteur die Möglichkeit, die Merkmale der gewünschten Anwendung komfortabel über eine grafische Benutzeroberfläche statt etwa durch die Angabe einer Grammatik (vgl. etwa Abschnitt 5.7.1) auszuwählen. Für jede zu konkretisierende Komponente wird wie beschrieben eine Oberfläche aufgebaut. Die `FeatureGUI` benötigt dazu Informationen, welcher Komponententyp konkretisiert werden soll, wie oft dieser Komponententyp eingesetzt werden soll und eine Liste mit allen Merkmalen, die für diesen Komponententyp zur Auswahl stehen. Ein Komponententyp kann einmal (mode=`one`) oder mehrfach (`more`) vorkommen oder optional (`optional`) sein. Es ist auch möglich, dass eine beliebige Zahl von Komponenten des Types eingesetzt werden kann (`any`). In letzterem Fall kann im Gegensatz zum Modus `more` auch *keine* Komponente eingesetzt werden (vgl. Abschnitt 6.3.2.1).

Für jedes Merkmal wird eine Karteikarte mit einem `RadioButton` für jeden möglichen Wert für die Auswahl zusammengestellt. In der Prototypimplementation ist die Reihenfolge der Karteikarten ebenso wie die der Merkmalswerte zufällig. Es liegen hierbei keine Prioritäten für besonders wichtige Features, bzw. deren Werte vor. Der Konstrukteur hat aber die Möglichkeit, auf jede beliebige Karteikarte zuzugreifen. Somit kann er die Reihenfolge der Merkmalsauswahl selbst bestimmen. Ein Merkmal kann beliebig viele Werte, mindestens aber einen anbieten. Zusätzlich wird für jedes Merkmal als Grundeinstellung ein `RadioButton` so belegt, dass auf Wunsch dieses Merkmal bei der Komponentenauswahl nicht berücksichtigt wird. Nach jeder Auswahl eines Merkmals werden alle Buttons deaktiviert, die Merkmalswerte repräsentieren, die von Komponenten angeboten werden, die die bis dahin festgelegten Kriterien nicht (mehr) erfüllen (vgl. a. Abschnitt 6.3.4).

6.3.3 ComponentPool

Der im Rahmen dieser Arbeit durchgeführte Entwurf eines objektorientierten Respository wurde in [Awizen, 2000] zunächst generisch implementiert und dann als Grundlage der Persistierung einzelner Komponenten im Sinne eines vom Typmanagements verwaltbaren »Pools« bereitgestellt. Auf eine detaillier-

te Bescheibung der Implementation soll hier verzichtet werden, da sie keine für den vorgestellten Ansatz zur Softwarekonstruktion spezifischen Probleme umfasst. Die `ComponentPool`-Komponente des Generators abstrahiert den Repository-Zugriff, sodass beispielsweise auch eine manuelle Eintragung oder Dateisystem-basierte Auswahl erfolgen kann.

Der Komponentenpool stellt dem Konstrukteur bei der Verwendung des Typmanager-Explorer (vgl. Abschnitt 6.2) eine Übersicht aller registrierten Komponententypen über alle Typdomänen inklusive deren Typ- sowie Komponenten-ID zur Verfügung. Als Zusatzinformation wird ausserdem der zu der entsprechenden Typbeschreibung korrespondierende »Display Name« [23] angegeben, der die instanzbezogene Komponentenbezeichnung – wie beispielsweise den Namen der Komponentenimplementation – darstellt. Während der Zugriff auf den Komponentenpool über das von der Generatorkomponente `ComponentPool` verwendete API nur indirekt möglich ist, indem zunächst auf die Komponententabelle mittels `getComponentTable()` zugegriffen und anschließend die zu den Einträgen korrespondierenden Display Names aus der Klasse `XMLtoTypehierarchy` gelesen werden, stellt der *Type Manager Explorer* diese Zusatzfunktion über »Extras« durch den Untermenüpunkt »Component Pool« direkt zur Verfügung. Der `ComponentPool` ist damit im Grunde eine Art *Broker*, der alle Komponentendeskriptoren einliest und eine Schnittstelle anbietet, um Informationen über die vorhandenen Komponenten abzufragen.

Der `ComponentPool` bietet dabei die Möglichkeit, Komponenten eines Types herauszufiltern, die bestimmte Merkmale erfüllen.[24] Der `ComponentPool` erzeugt hierfür eine Instanz der Klasse `ComponentList` für jeden Komponententyp. Es wird wie im gesamten Typmanagement der DOM-Standard [W3C, 2000c] zur programminternen Repräsentation und Bearbeitung der Komponentendeskriptoren verwendet. Die Informationen über eine Komponente werden in einer Instanz der Klasse `Component` abgelegt. Die Klasse `Component` bietet die Möglichkeit, einige Informationen über die Komponente zu erfragen, wie den Namen der Komponente oder ihren Komponententyp. Über eine weitere Methode kann untersucht werden, ob eine Komponente bestimmte Merkmale erfüllt. Es werden nur die Informationen, die die Komponente allgemein beschreiben und die Auswahl von Kompo-

[23] vgl. a. die EJB- beziehungsweise CCM-Spezifikationen [Sun, 2000],[OMG, 1999]

[24] vgl. a. das Konzept eines *Traders* [Müller et al., 1994]

nenten direkt betreffen, berücksichtigt. Technische Details konkreter Instanzen, etwa die Methoden der syntaktischen Ebene, werden dabei nicht beachtet, um die gewünschte Abstraktion von verschiedenen Komponententechnologien und den fachlich orientierte Blickwinkel des Softwarekonstrukteurs aufrecht zu erhalten.

Alle Anfragen an den `ComponentPool` werden weitergeleitet an die `ComponentList` für den jeweils gesuchten Komponententyp. Das Ergebnis der Suchanfragen sind Referenzen auf Instanzen der Klasse `Component`, die die genannten Kriterien erfüllen. Es exisieren derzeit keine übergeordneten Entscheidungskriterien, wie eine einzelne Komponente bestimmt wird, wenn mehrere Komponenten die Suchkriterien erfüllen. Es wird einfach die erste Komponente aus der Ergebnisliste ausgewählt. Der `ComponentPool` bietet zudem die Möglichkeit, eine Liste aller möglichen Merkmale für einen Komponententyp zu erhalten. Hierbei können auch bereits ausgewählte Merkmale als Parameter übergeben werden, um das Ergebnis einer Anfrage weiter einzuschränken. Dies wird benötigt, um bei der Komponentenauswahl nur gültige Merkmalskombinationen für eine Komponente eines bestimmten Komponententyps zuzulassen. Wie in Abschnitt 6.3.2.2 diskutiert wurde, verhindert dies nicht, dass für einen anderen später zu spezifizierenden Komponententyp keine passende Komponente gefunden werden kann.

6.3.4 Zusammenspiel der Komponenten bei der Merkmalsauswahl

Der `Generator` bestimmt den jeweils nächsten zu konkretisierenden Komponententyp anhand der vorliegenden Architekturbeschreibung. Im Anschluss wird der `ComponentPool` befragt, welche Merkmale für diesen Komponententyp zur Auswahl stehen. Um die bisher direkt und indirekt ausgewählten Merkmale (vgl. Abschnitt 5.10.2) zu berücksichtigen, muss bestimmt werden, welche Merkmale für diesen Komponententyp signifikant sind. Die interaktive Merkmalsauswahl soll nur die Merkmale von Komponenten umfassen, die von Komponenten angeboten werden, die *alle* signifikanten Merkmale implementieren. Eine Liste mit noch freien Merkmalen (also bis dahin ohne Auswahl der Wertbelegung) wird zusammen mit den Informationen über den Komponententyp und dem Modus, d.h. wie oft der Komponententyp vorkommen darf, an die `FeatureGUI` übergeben. Die-

se baut eine diesen Informationen gemäß gestaltete Oberfläche auf.[25]

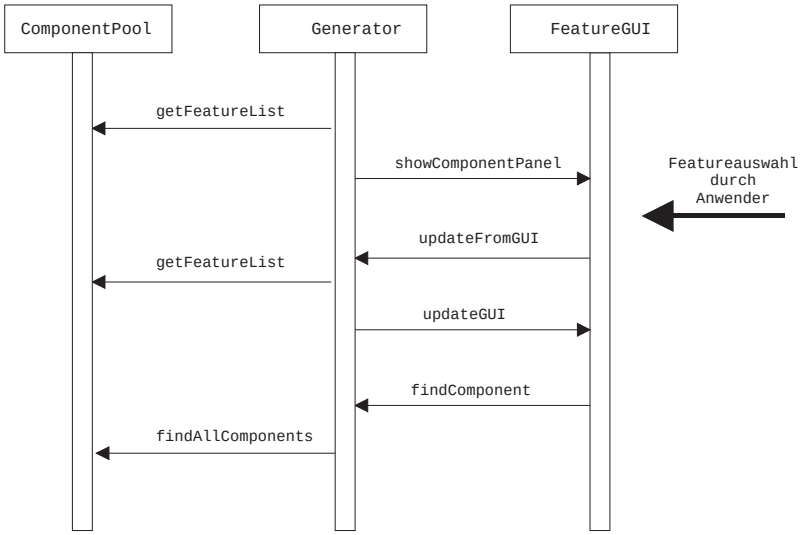


Abbildung 6.15 *Ablauf der Merkmalsauswahl*

Wenn ein Merkmal durch den Anwender ausgewählt wird, werden der assoziierte Name und Wert in eine Liste der ausgewählten Merkmale des bearbeiteten Komponententyps eingetragen. Der Anwender kann die Wahl eines Merkmals zurücknehmen, indem er den Standardwert[26] auswählt. In diesem Fall wird das vorher ausgewählte Merkmals-Name-Wert-Paar aus der Liste der ausgewählten Merkmale entfernt. Die aktualisierte Liste wird an den `Generator` weitergegeben. Der `Generator` schickt eine Anfrage an den `ComponentPool` nach einer Liste aller freien Merkmale, die von Komponenten angeboten werden, die alle interaktiv ausgewählten ebenso wie die für diesen Komponententypen siginifikanten Merkmale implementieren. Das Ergebnis der Anfrage wird an die `FeatureGUI` weitergegeben. Dort bleiben nur die Buttons aktiv, die Merkmale repräsentieren, die in dieser Liste vorkommen. Auch Standardwerte können weiterhin ausgewählt werden. Alle anderen Buttons werden deakti-

[25] D.h. insbesondere, dass sich der Generator dem Konstrukteur durchaus unterschiedlich präsentiert, je nach Art der zu erstellenden Anwendung und der dafür zur Verfügung stehenden Komponenten. Allerdings bleibt diese Variablität stets in einen wiedererkennbaren Gesamtrahmen eingebunden, sodass der Konstrukteur nicht einem kontinuierlichen Umdenken oder Lernprozess in der Nutzung ausgesetzt wird.

[26] der ja gerade besagt, dass ein Merkmal bei der Auswahl einer Komponente eines bestimmten Types nicht berücksichtigt werden muss

viert, sodass keine ungültigen Feature-Kombinationen für diesen Typ ausgewählt werden können. Diese Prozedur wiederholt sich, bis der Konstrukteur die Auswahl einer konkreten Komponente startet.

Eine Komponente vom Typ Funktionseinheit, die die Grundrechenarten anbietet (vgl. Abschnitt 5.12), hat zum Beispiel das Merkmal `Funktionsart=Grundrechenarten`. Wenn dieses Merkmal ausgewählt wurde, sind nur noch die Merkmale gültig, die von Komponenten angeboten werden, die auch dieses Merkmal anbieten, wovon es in der Beispielimplementation der Taschenrechnerdomäne zwei Komponenten gibt. Diese unterscheiden sich in der Art der Notation. Der Konstrukteur hat nur noch die Wahl zwischen den Merkmalswerten `RPN` und `Standard` für das Merkmal `Notation`. Merkmale, wie die der trigonometrischen Funktionseinheiten(etwa `Methodenname=Cosinus`) stehen erst wieder zur Auswahl, wenn das Merkmal `Funktionsart=Grundrechenarten` zurückgenommen wurde, d.h. der Standardwert für den Merkmalsbezeichner `Funktionsart` ausgewählt wurde. Auf diese Weise wird eine nicht zu erfüllende Merkmalskombination im Vorwege verhindert, etwa eben die Auswahl `Methodenname=Cosinus` und `Funktionsart=Grundrechenarten`, die keine einzelne Komponente der Beispielimplementation erfüllen kann.

6.3.5 Aufbau der Anwendungsstruktur

Wenn die Komponenten, die für die gewünschte Anwendung benötigt werden, ausgewählt wurden, muss die notwendige Anwendungsstruktur abgeleitet werden. Dies erfolgt, indem aus einer Liste der Deskriptoren der ausgewählten Komponenten und der DTD, die die möglichen Anwendungsstrukturen für die Domäne beschreibt, ein für diese DTD gültiges XML-Dokument generiert wird, bei dem alle Komponenten als Elemente auftreten. Jede ausgewählte Komponente soll zu einem Element im XML-Dokument werden, wobei wie erwähnt der Name des Komponententyps dem jeweiligen Elementnamen entspricht. Diese Beschreibung einer konkreten Anwendungsstruktur dient dem `AppBuilder` als Grundlage, um die gewünschte Anwendung zu konstruieren.

Es ist also das Problem zu lösen, aus einer bestimmten Menge von Elementen und einer DTD ein gültiges XML-Dokument zusammenzustellen, was einer *Umkehrung* des üblichen Umgangs mit XML-Dokumenten bzw. deren Parsing entspricht. Das Ergebnis ist dabei nicht notwendigerweise eindeutig. Eventu-

ell können mehrere gültige XML-Dokumente gefunden werden. Der `XMLDocBuilder` liefert nur eines dieser möglichen XML-Dokumente, wobei einige Einschränkungen auf die mögliche Vielfalt der DTD vozunehmen sind, auf die später eingegangen wird. Je spezifischer die Anwendungsstruktur und damit die DTD ist, desto eindeutiger kann die gewünschte Struktur bestimmt werden. Abschnitt 6.3.5.3 verdeutlicht, wie die Ausdrucksmöglichkeiten der DTD für den Einsatz als Architekturbeschreibung eingeschränkt werden müssen, damit der Generator diese verwenden kann.

6.3.5.1 Der XML-Parser

Normalerweise wird ein bestehendes XML-Dokument gegen eine DTD geprüft. Im Falle der Komponente `XMLDocBuilder` wird der umgekehrte Weg eingeschlagen und stattdessen die DTD als Grundlage genommen, um ein XML-Dokument erst *aufzubauen*. Es wird somit gewissernmassen ein *konstruktives Parsing* vorgenommen. Um ein gültiges XML-Dokument zusammenstellen zu können, ist es dann notwendig, überhaupt eine DTD auslesen zu können. Diese Funktionalität ist noch nicht durch das World Wide Web Consortium (W3C) standarisiert. Erst die Spezifikation des DOM-Modells Level 3 soll diese Funktionalität standarisieren. Zum Zeitpunkt der Erstellung der vorliegend dokumentierten Prototypen musste somit entschieden werden, einen eigenen Parser zu implementieten oder existierende Vorabversionen des zu erwarteneden W3C-Standards zu nutzen. Die Wahl fiel auf den Java-XML-Parser von IBM in der Version 2.0.0. Diese Version beinhaltet eine Klasse `DTD`, die die notwendigen Abfragen ermöglicht, welche Elemente als jeweils nächstes in ein Dokument eingesetzt werden können, wenn dennoch ein für diese DTD gültiges XML-Dokument entstehen soll.[27] Diese Entscheidung erschien im Sinne einer Arbeitserleichterung der praktischen Arbeiten im Rahmen dieser Arbeit sinnvoll, da ja der Nachweis der konzeptuellen Tragfähigkeit durch Implementation und nicht deren vollständige Produktunabhängigkeit im Vordergrund stand.

6.3.5.2 Funktionsweise des XMLDocBuilder

Der XML-Baum wird von der Wurzel aus in die Tiefe erstellt. Der Baum repräsentiert dabei das zu erstellende XML-Dokument.

[27] Achtung: Neuere Versionen dieses Parsers ab Version 3 bieten diese Möglichkeit derzeit nicht mehr. IBM hat diese Fähigkeit auf Grund der noch nicht abgeschlossenen Standardisierung unsinnigerweise entfernt.

Es wird im weiteren von Elementen gesprochen, die eingesetzt werden, wobei hiermit die Elemente gemeint sind, die die ausgewählten Komponenten repräsentieren. Der `XMLDocBuilder` erstellt zuerst das Root-Element. Das Root-Element trägt als Bezeichnung immer den Namen der jeweils betrachteten Domäne bzw. des betrachteten Fachkonzepts der Domäne (vgl. Abschnitt 6.4.3). Mit diesem Element wird die Methode `insertElements(Element elem, String lastName)` aufgerufen. Die Bedeutung der einzelnen Parameter wird in der näheren Erläuterung der Funktionsweise des Auslesens der DTD deutlich werden. Die Methode `insertElements` ruft sich selbst rekursiv auf, wenn Elemente an den jeweils bereits zusammengefügten Teil des Dokumentes angehängt werden.

Die Klasse `DTD`, die der gewählte XML-Parser zur Verfügung stellt, bietet die Methode `getAppendableElements(Element el, Hashtable tab)` an. Diese Methode liefert eine Liste mit Elementen, die an einer Stelle in ein Dokument eingesetzt werden können und dabei der DTD gehorchen. Hierfür muss zuerst die übergebene `Hashtable` mittels der Methode der Klasse `DTD` `prepareTable(String elementname)` präpariert werden. Diese liefert eine Liste mit Elementen, die überhaupt bei einem Element mit dem gerade betrachteten Namen eingesetzt werden können. Die Methode `getInsertable` filtert die Elemente heraus, die für einen bestimmten Punkt im Dokument gültig wären. Das Ergebnis steht wieder in der übergebenen `Hashtable`. Dieses Ergebnis wird im `XMLDocBuilder` mit der Methode `getNextElements(Hashtable ht)` zur einfacheren weiteren Bearbeitung in eine Liste mit den Namen der möglichen Elemente eingetragen.

Ausser Elementnamen als Rückgabewerte gibt es die Möglichkeit, dass ein mögliches Ende eines Zweiges angezeigt wird (wird als `EOC` bezeichnet oder ein Fehler aufgetreten ist (intern `EOF` genannt. Ein Fehler tritt auf, wenn der bisherige Teilbaum nicht der DTD entspricht. Der Vorgang wird in diesem Fall sofort abgebrochen. Ein mögliches Ende eines Zweiges steht immer am Ende der Liste. Der `XMLDocBuilder` geht zuerst alle anderen Möglichkeiten durch und versucht Elemente einzusetzen. Wenn aber keines der möglichen Elemente vorhanden ist, weil es sich zum Beispiel um optionale Elemente handelt und die entspechenden Komponenten nicht ausgewählt wurden, und ein mögliches Ende als einziges übrigbleibt, wird dieser Teilbaum als abgeschlossen angesehen. Es wird dann untersucht, ob in der Ebene darüber noch Elemente eingesetzt werden können. Das XML-Dokument ist fertig erstellt, wenn beim Root-Element ein

mögliches Ende erreicht wurde, wobei zu beachten ist, dass vorher schon versucht wurde alle anderen möglichen Elemente einzusetzen.

Es wird zuerst untersucht, welche Elemente beim Root-Element eingesetzt werden können. Das erste mögliche Element aus der zusammengestellten Liste wird eingesetzt, zumindest wenn mindestens eine diesem Element entsprechende Komponente in der Spezifikationsphase der Generierung ausgesucht wurde. Es ist derzeit kein weiteres Backtracking vorgesehen, das es erlauben würde mehrere Möglichkeiten auszuprobieren. Nachdem an das Root-Element ein Element angehängt wurde, wird das hinzufügte Element nach einsetzbaren Elementen untersucht und auch eines dieser Elemente hinzugefügt.

Es gibt hierbei zwei Möglichkeiten, in welcher Rolle neue Elemente eingesetzt werden können:

- Sohn – Der Sohn wird an den bisherigen Baum beim Vaterknoten angehängt. Der Sohn wird rekursiv nach dessen möglichen Söhnen untersucht, bis der Teilbaum abgeschlossen ist. Dieser Fall ist simpel, da einfach ein mögliches Element aus der Liste an den bisherigen Baum angehängt wird und die Methode rekursiv mit dem neuen Element als Parameter aufgerufen wird.

- Element in der gleichen Ebene – Es muss untersucht werden, welche Elemente an einer Stelle eingesetzt werden können, wenn schon ein Element eingesetzt wurde. Hierbei muss verhindert werden, dass das gleiche Element noch einmal eingesetzt wird. Wenn beliebig viele Elemente eines Typs eingesetzt werden können, setzt der `XMLDocBuilder` alle vorhandenen Elemente ein und fragt dann nach, welche Elemente an der Stelle möglich sind, wenn das letzte Element, das ja beliebig oft vorkommen durfte, eingesetzt wurde. Das gleiche Element darf nach den Regeln der DTD nochmal eingesetzt werden. Dementsprechend muss hier die Entstehung eines Zyklus abgefangen werden, in dem das gleiche Element ein weiteres Mal eingesetzt würde. Daher wird der Name des zuletzt eingesetzten Elementes mit übergeben und dieses Element wird nicht eingesetzt, sondern das nächste Element aus der Liste der möglichen Elemente. Wenn kein anderes Element eingesetzt werden kann und ein Ende des Teilbaumes möglich ist, wird der Vaterknoten nach weiteren möglichen Elementen untersucht.

Illustration der Arbeitweise In Abbildung 6.16 ist der Input und der Output für ein einfaches Beispiel zu sehen. Die zu Grunde liegende DTD fordert ein Root-Element `A`. An dieses muss ein `B` und ein `C` oder ein `D` (Entweder-Oder-Entscheidung) angehängt werden. Bei `D` ist ein optionales Element `E` möglich. Es stehen die Elemente `A, B, D` zur Verfügung. Aus der DTD und den Elementen muss der `XMLDocBuilder` ein gültiges XML-Dokument erstellen. In der Tabelle 6.1 wird illustriert wie der `XMLDocBuilder` dabei zu seinem Ergebnis kommt. Es ist gezeigt, in welcher Reihenfolge, bei welchem Element (*Vater*) die einzelnen Elemente eingesetzt werden. Die *möglichen Elemente* bezeichnen Elemente, die an den *Vater* angehängt werden können und dabei zu einem gültigen XML-Dokument führen können. Die Auswahlmöglichkeit wird durch die zur Verfügung stehenden Elemente eingeschränkt. Der `XMLDocBuilder` setzt zum Beispiel Element `D` an Position 4 ein, weil kein Element `C` vorhanden ist. `EOC` bezeichnet immer ein gültiges Blatt eines Zweiges, d.h., dass an dieses Element kein weiteres Element angehängt werden muss.

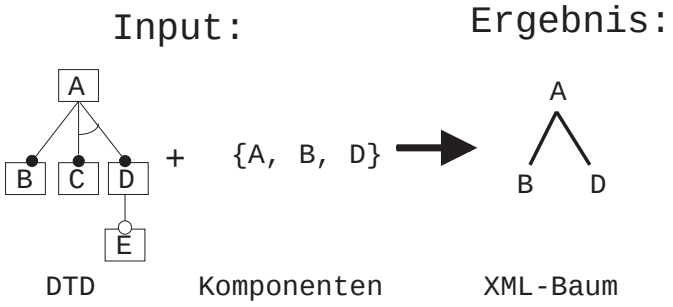


Abbildung 6.16
Beispiel wie der XML-DocBuilder arbeitet

Nr.	Vater	mögliche Elemente	eingesetzt
1.	-	A	A
2.	A	B	B
3.	B	EOC	-
4.	A	C,D	D
5.	D	E,EOC	-
6.	A	EOC	-

Tabelle 6.1
Ablauf der Erstellung eines XML-Baumes

Wenn das XML-Dokument zusammengestellt ist, muss es noch in eine Datei geschrieben werden. Hierfür wird auf komfortable Erweiterungen des verwendeteten IBM-Parsers, die nicht

der DOM-API entsprechen, zurückgegriffen. Sie ermöglichen durch die Klasse `TXDocument`, die die Möglichkeiten von `Document` erweitert, auf eine Methode zu zugreifen, die diesen Vorgang erheblich erleichtert.

6.3.5.3 Einschränkungen des XMLDocBuilder

Das Root-Element kennzeichnet mit seinem Namen im Sinne eines fachlichen Domänen*konzepts* (vgl. Abschnitt 5.3) die Art der gewünschten Anwendung. Dementsprechend exisiert genau eines. Die anderen Elemente in der DTD entsprechen den Namen von Komponententypen. Es ist nicht sinnvoll, ein Root-Element mit mehr als einem Sohn zuzulassen. Die gewählte Anwendungsstruktur sieht vor, dass nur benachbarte Komponenten miteinander kommunizieren können. Wenn an der Wurzel mehr als ein Sohn-Element eingesetzt würde, können die Teilbäume nicht untereinander kommunizieren, da sie keinen benachbarten Knoten besitzen. Es sollte an dieser kritischen Stelle also stets eine für die jeweilge Anwendung grundlegende Komponente mit der Rolle eines »Verteilers« (Dispatcher, Koordinator, Controller etc.) vorgesehen werden, der zwischen den Teilbereichen der Anwendung die Kommunikation übernimmt. Im Falle des Taschenrechners ist es dies etwa der `Aktive_Speicher` und in der Hierarchie entsprechend über allen anderen an Berechnungen beteiligten Komponenten steht. Ansonsten ist es auch möglich, eine Komponente, die nicht an der eigentlichen Funktionalität der Anwendung beteiligt ist, sondern zum Beispiel für die Visualisierung von Komponenten zuständig ist, d.h. »GUISlots« anbietet (s. Abschnitt 6.3.6.5), in der Hierarchie noch über dieser »Verteiler« -Komponente anzuordnen, wie dies auch bei der Beispieldomäne »Taschenrechner« (s. Abschnitt 5.12 geschehen ist.

Wenn ein Komponententyp optional ist, wird eine entspechende Komponente nur eingesetzt, wenn auch eine in der Spezifikationsphase ausgewählt wurde. Wenn ein Element in der DTD optional ist, erscheint das Element zwar in der Liste der möglichen Elemente, der `XMLDocBuilder` springt aber zum nächsten möglichen Element, wenn keine entsprechende Komponente in der Liste der ausgewählten Komponenten vorhanden ist. Dies passiert ebenso, wenn ein Element beliebig oft, also auch kein einziges Mal, eingesetzt werden kann. Analog verhält es sich mit getroffenen Entweder-Oder-Entscheidungen aus der Spezifikationsphase. Wenn die Wahl zwischen zwei möglichen Komponententypen bestand, dann ist nur genau eine dieser Komponenten ausgewählt worden. Der `XMLDocBuilder` kann an die-

ser Stelle dadurch auch nur eine dieser Komponenten einsetzen. Dies bedeutet jedoch auch, dass Komponententypen, die bei Entweder-Oder-Entscheidungen beteiligt sind, nicht in Teilbäumen der anderen beteiligten Komponententypen auftreten dürfen. Seien Element `A` und `B` entsprechende mögliche Werte von Entweder-Oder-Entscheidungen. Wenn `A` ausgewählt wurde, darf `B` nicht in dem Teilbaum von `A` vorkommen. Element `A` selbst darf an beliebig vielen Stellen im eigenen Teilbaum vorkommen, ebenso wie `B` in dem seinigen. Wenn `B` in dem Teilbaum von `A` vorkäme, hätte der `XMLDocBuilder` eine zufällige Entscheidung bei der Entweder-Oder-Wahl zu treffen. Es wären an dieser Stelle Elemente für beide Möglichkeiten der Entweder-Oder-Entscheidung vorhanden, in diesem Fall sowohl `A`, als auch `B`) und der Generator müsste sich für eine davon entscheiden. Die Entscheidung sollte aber durch die ausgewählten Komponenten eindeutig zu bestimmen sein.

Für manche Komponententypen wurden möglicherweise mehrere Komponenten ausgewählt. Daher muss der `XMLDocBuilder` ein XML-Element für jede konkretisierte Komponente eines Typen einsetzen, wenn dies nach der DTD gültig ist. Da die DTD und die dazugehörige Komponentenauswahl aufeinander abgestimmt sind, ist dies immer der Fall. Es ist allerdings nicht möglich, Komponente `A1` an einer Stelle und Komponente `A2` des gleichen Types an einer anderen Stelle des XML-Dokumentes einzusetzen. Es werden immer `A1` und `A2` an beiden Stellen eingesetzt. Dies ist nicht mit der Diskussion der Vermeidung einer instanzlichen Mehrfachverwendung in Abschnitt 2.3.2.1 zu verwechseln, da zu diesem Zeitpunkt hier noch keine endgültige Entscheidung für eine der konkreten Komponenten gefallen ist. In der derzeitigen Implementation wird immer die gleiche Anzahl von Elementen eines Typs an jeder Stelle, an der er vorkommen kann, eingesetzt. Diese Aussage betrifft insbesondere auch Elemente, die an einigen Stellen vorkommen können, aber nicht müssen, d.h. die entweder optional sind oder beliebig oft, also auch kein mal, vorkommen können. Der Generator überprüft in diesen Fällen, ob eine oder mehrere Komponenten des Typs, den dieses Element repräsentiert, in der Spezifikationsphase ausgewählt wurde und setzt diese gegebenenfalls an jeder möglichen Stelle ein. Es ist nicht möglich, einen Komponententyp an einer Stelle einzusetzen und an einer anderen Stelle, wo dieser ebenfalls möglich, aber nicht notwendig ist, nicht. Diese Entscheidung hängt einzig und allein davon ab, ob eine oder mehrere entsprechende Komponenten im Spezifikationsvorgang ausgewählt wurden.

Der implementierte Algorithmus kann nur einfache Zyklen verhindern (vgl. a. Abschnitt 2.3.2.1), da der `XMLDocBuilder`, wenn ein einziges Element beliebig oft eingesetzt werden kann, nur den Namen des zuletzt eingesetzten Elementes vermerkt und dieses Element nicht ein zweites Mal an der gleichen Stelle im XML-Baum einsetzen würde. Es ist nicht möglich größere Schleifen zu erkennen. Wenn an einer Position die Teilbäume der Elemente `A` und `B` beliebig oft eingesetzt werden können (in DTD-Notation: `(A|B)*`), könnte der `XMLDocBuilder` den Teilbaum von `A` einsetzen, wodurch sich aber die Möglichkeiten, welche Elemente als Nächstes eingesetzt werden können, nicht ändern würden. Für den `XMLDocBuilder` bestünde die gleiche Situation wie zuvor, d.h. er könnte wieder die Teilbäume der Elemente `A` und `B` beliebig oft einsetzen. Diese Art der Zyklenbildung sollte gemäß des Komponentenmodells aus Kapitel 2 verhindert werden und daher auf einfache Zyklen abgebildet werden.

Viele Zyklen, die in einer DTD beschrieben werden können, sind für eine konkrete Anwendungsarchitektur nicht notwendig. Viele Fälle lassen sich durch einfache Umformungen ersetzen. Eine Spezifikation wie `(a,b)*` kann umgeformt werden in `(a*,b*)`, wobei die Unterschiede durch den Einsatzzweck als Anwendungsstruktur unbedeutend werden. Während es für einige Anwendungsfälle wichtig ist, ob `(ababab)` oder `(aaabbb)` ausgewählt wurde, ist dies für den Generator nicht der Fall. Alle eingesetzten Komponenten werden auf die gleiche Weise mit ihrem Vater und ihren Söhnen in der Anwendungsstruktur verknüpft. Die Reihenfolge in einer Ebene stellt dabei kein Unterscheidungskriterium dar.

Eine andere Möglichkeit wäre es, eine weitere Hierarchieebene einzuführen und `(c*)` mit `c=(a*,b*)` zu verwenden. `c` könnte durch eine *abstrakte* Schnittstelle (vgl. Abschnitt 5.12.6) gefüllt werden, sodass keine Komponente dieses Types wirklich in die Anwendung eingesetzt wird. Das gleiche Prinzip kann auch bei Oder-Spezifikationen (zum Beispiel `(a|b)*` wird zu `(a*|b*)`) eingesetzt werden. Es ist bei den Umformungen wichtig, mögliche mehrstufige Zyklen zu verhindern und auf einfache Zyklen, die der `XMLDocBuilder` erkennen kann, zurückzuführen.

Eine andere wichtige Einschränkung ist, dass der `XMLDocBuilder` nur untersuchen kann, welche Elemente an einer Stelle eingesetzt werden können. Die Klasse `DTD` liefert keine Informationen, wie oft ein Element eingesetzt werden kann oder ob es sich um ein optionales Element handelt. Ist keine Priorität von Elementen vorgegeben, wird zufällig eines der möglichen

Elemente ausgewählt und eingesetzt. Deshalb sollte die DTD so ausgelegt sein, dass die Reihenfolge der Einsetzung von möglichen Elementen an einer Stelle möglichst keinen Einfluss auf das Ergebnis hat, was aber durch die hierarchische Softwaresicht der zu Grunde gelegten Merkmalsmodelle begünstigt wird.

6.3.6 Quellcode-Generierung für das Anwendungsgerüst

Der `AppBuilder` hat die Aufgabe, aus den ausgewählten Komponenten und der generierten Anwendungsstruktur, die gewünschte Anwendung zu konstruieren. Es wird im Falle des implementieren Prototypen dabei der Java-Quellcode für eine Rahmenanwendung generiert, die die ausgewählten Komponenten instanziiert und miteinander durch eine entsprechende Ereignisregistrierung verknüpft. Ausgangsbasis dieses Prozesses ist das XML-Dokument, das der `XMLDocBuilder` zusammengestellt hat. Das XML-Dokument enthält neben den Angaben über die Struktur der Anwendung, die in Form des XML-Baumes vorliegt, nur die URLs[28] zu den Schnittstellenbeschreibungen der benötigten Komponenten. Im Gegensatz zum `ComponentPool` werden hierbei die implementationstechnischen bzw. syntaktischen, also technologieabhängigen Informationen ausgelesen (vgl. Abschnitt 6.3.3). Hierbei handelt es sich um Informationen über die vorgesehenen Verknüpfungspunkte der Komponenten sowie Informationen, die für die Instanziierung und für die Post-Konfiguration benötigt werden. Die Anzahl einzelner Komponententypen kann sich noch verändern, da eine Komponente in ihrem Deskriptor Angaben über benötigte (weitere) Komponenten enthalten kann. Der `AppBuilder` ist auch für die Anleitung des Konstrukteurs in der Post-Konfiguration zuständig.

6.3.6.1 Individuelle Informationen über eingesetzte Komponenten

Der `AppBuilder` instanziiert für jede Komponente, die in die Anwendung eingesetzt werden soll, ein Objekt der Klasse `Gen-`

[28] Die durch das WWW etablierten Uniform Resource Locators (URLs) wurden zur Referenzierung konkreter Komponenten(instanzen) gewählt, um eine möglichst allgemeingültige Repräsentation zu verwenden, die auf eine Vielzahl von Referenzierungsarten unterschiedlicher Komponententechnologien abgebildet werden kann, z.B. CORBA Interface-Repository, EJB-LDAP Namendienste oder Microsoft Component Catalogs.

Component. Hierfür wird genauso wie bei der Klasse Component in der Spezifikationsphase, die Schnittstelle einer Komponente ausgelesen, wobei diesmal nicht die Spezifikationsteile, die für die Auswahl einer Komponente von Bedeutung sind, sondern die Spezifikationen, die bei der Komposition der Komponente eine Rolle spielen, ausgewertet werden. Wenn eine Komponente, wie zum Beispiel ein Button, in einer Anwendung mehrfach vorkommt, werden auch mehrere GenComponent-Instanzen für diese Komponente erstellt. Dies ist notwendig, da jede Instanz einen zugeordneten Variablennamen mit sich führt. Der Variablenname wird benötigt, um im generierten Quellcode eine Komponente ansprechen und damit auch mit anderen Komponenten verknüpfen zu können. Der Variablenname wird vom AppBuilder automatisch vergeben und besteht aus dem Komponententypnamen und einer Zahl, die bei jedem Vorkommen einer Komponente des jeweiligen Typs erhöht wird. Auch die Daten für die Post-Konfiguration werden über die GenComponent-Instanzen verwaltet, also insbesondere die instanzlichen Attribute (Properties) von Komponenten. Da auch die Post-Konfiguration der Komponenten zur besseren Fehlervermeidung durch den Konstrukteur dem Policy-Management, also dem eventuell notwendigen automatischen Angleich entsprechender Komponenteneigenschaften (vgl. Abschnitt 3.10), unterworfen ist, wickeln die GenComponents auch die Kommunikation mit dem Policy-Manager ab.[29]

Neben den Informationen über die Komponente selbst werden individuelle Informationen über benötigte Komponenten mit Hilfe der GenComponents verwaltet. Auf diese Weise kann eine Komponente wie etwa die Grundrecheneinheit der Beispieldomäne, mit mehreren Buttons verknüpft werden, auch wenn im XML-Dokument nur ein Button als Kindknoten angegeben ist. Die Klasse GenComponent bietet hierfür die Möglichkeit für jeden Komponententyp für in der Anwendungshierarchie unter ihr liegende Komponenten abzufragen, wieviele Komponenten eines bestimmten Types hinzugefügt werden sollen. Der AppBuilder setzt dann die entsprechende Menge an Komponenten dieses Typs ein. Es kann ebenfalls nach vorhandenen Beschreibungen für eine oder mehrere Komponenten eines bestimmten Typs gefragt werden. Diese Beschreibungen betreffen individuelle Konfigurationseinstellungen, wie mögliche Beschriftungen für einen

[29] Der Fall, dass selbst-überwachte Komponenten [Griffel, 1998], also solche, die selbstständig am Policymanagement teilnehmen (vgl. a. [Tetau, 1999]), ist selbstverständlich möglich. Um generisch zu bleiben, sollte jedoch nicht davon ausgegangen werden, dass jeder Komponentenentwickler entsprechenden Code vorgesehen hat.

`Button` (vgl. Property-Deskriptoren des CCM, Abschnitt 6.22). Diese Beschreibungen sind Vorschläge und können durch den Anwender in der Post-Konfiguration geändert werden.

Einer der Gründe, hierbei nicht direkt auf die Komponentenbeschreibungen im `ComponentPool` zurückzugreifen, liegt im Offenhalten der Möglichkeit auch ein nicht instanzorientiertes Typmanagement zuzulassen, also eines das im Gegensatz zum in Kapitel 4 vorgestellten einzelne Instanzen eines Komponententyp nicht unterscheiden kann. Zudem liegt die Aufgabe des `ComponentPools` rein in der Auswahl der geeigneten Komponenten. Die *Verwendung* der verwalteten Komponenten ist hiervon klar zu trennen, zumal wie oben beschrieben jeweils unterschiedliche Bereiche der Deskriptoren des Typmodells ausgewertet werden. Sowohl die Klasse `Component`, als auch die Klasse `GenComponent` lesen zwar allgemeine Informationen aus den Komponenten-Deskriptoren, wie etwa den Namen der Komponente. Aber während der `ComponentPool` allgemeine Beschreibungen der Komponente in Form der Merkmale einer Komponente verwaltet, konzentriert sich die `GenComponent` auf konkrete (technische) Ausprägungen, die die Komposition betreffen.

6.3.6.2 Funktionsweise des AppBuilder

Der `AppBuilder` generiert eine Rahmenanwendung, die im Falle visueller Anwendungen ein Fenster für die Applikation bereitstellt und die einzelnen Komponenten miteinander verküpft, sowie individuelle Konfigurationseinstellungen vornimmt. Der zu generierende Quellcode lässt sich in zwei Teile aufteilen:

- dem allgemeinen Quellcode, der für alle Anwendungen gleich ist, und
- dem notwendigen, individuellen Quellcode für die Assoziation jeder einzelnen Komponente mit der Anwendungsstruktur.

Es werden dabei mehrere Queellcode-Dateien erzeugt. Zwischen den allgemeinen Teilen des Quellcodes wird eine Methode aufgerufen, um den Baum, den das XML-Dokument und damit die Anwendungsstruktur bildet und der die einzusetzenden Komponenten bestimmt, abzuarbeiten. Das XML-Dokument wird von der Wurzel in die Tiefe abgearbeitet. Der Quellcode zur Einbindung jeder einzelnen Komponente wird erstellt. Es wird hierfür eine Methode mit einer Referenz auf das aktuelle Element im Baum, sowie Referenzen auf die dazugehörige Komponente

und die in der Hierarchie darüberliegende Komponente aufgerufen. Durch die Verwendung eines XML-Dokumentes zur Beschreibung der Struktur, ist sichergestellt, dass eine Komponente nur genau einen Vater, aber mehrere Söhne haben kann. Die aufgerufene Methode ruft sich rekursiv für alle Unterelemente des aktuellen Elementes auf und traversiert so den gesamten Baum. Der Quellcode zur Anbindung jeder Komponente an das Anwendungsgerüst umfasst immmer:

- **Instanziierung der Komponente**
  Hier ist eine *Factory* (vgl. [Gamma et al., 1994]) vorgesehen, die die Abstraktion von unterschiedlichen Instanziierungsmechanismen leistet. Derzeit wird die Instanziierung von Java-Objekten (Bean- und JMS-Komponenten), COM-Objekten (lokale, »surrogate« und MTS-Komponenten) sowie CORBA-Objekten unterstützt.[30]

- **Graphische Darstellung**
  Einige Komponenten benötigen eine graphische Darstellung. Eine der ausgewählten Komponenten sollte »Slots« anbieten, um diese anzuzeigen (s. Abschnitt 6.3.6.5). Es existieren natürlich auch Komponenten ohne graphische Darstellung (in der Regel die Mehrzahl).

- **Einstellen der Eigenschaften einer Komponente**
  Manchmal ist es wünschenswert, eine Komponente für den individuellen Einsatz zu konfigurieren. Es ist im Moment nur vorgesehen, mit Hilfe einer manuellen Post-Konfiguration und einem automatischen Policy-Management eine entsprechende Anpassung der zu generierenden Anwendung zu erlauben.

- **Verknüpfung der Komponenten untereinander**
  Es werden nur im Baum direkt benachbarte Komponenten gemäß der Informationen ihrer Typbeschreibungen miteinander verknüpft. Die Verknüpfungen können bidirektional erfolgen.

[30] Da die Instanziierung insbesondere auch entfernt erfolgen kann und entsprechend komplexe bzw. spezifische Mechanismen nicht in den generisch gehaltenen Generatorentwurf einfließen sollten, stellt auch die Factory wieder eine Kapselung dar, in diesem Falle externer Unterstützungsdienste für Aktualisierung und verteiltes Konfigurationsmanagement (entsprechende Dienste wurden im Rahmen dieser Arbeit in [Broschei, 2000][Lapok, 2000][Martens, 1999][Benthien, 1999] und [Häming, 1999] entwickelt).

Für alle eingesetzten Komponenten werden die benötigten Dateien (Code, Images, Deskriptoren etc.) in eine Liste eingetragen. Diese Liste wird später benötigt, um eine Archivdatei – das »Auslieferungsformat« einer komponentenbasierten Applikation – zusammenstellen zu können. Eine Komponente kann auch die URLs von zusätzlich benötigten Dateien in ihren Deskriptoren angeben (vgl. a. Abschnitt 6.2.1.2).

6.3.6.3 Verknüpfung von Komponenten

Die Komponenten der hier angesprochenen Beispiel-Implementation werden mit Hilfe von Java-Events verknüpft. Normalerweise muss hierfür nur der Listener bei der Quelle des Events eingetragen werden. `ActionEvent`s werden als ein Spezialfall behandelt, weil sie ein häufig eingesetztes Ereignis sind und häufig zwischen unterschiedlichen Quellen des Ereignisses unterschieden werden muss. Eine Komponente vom Typ `Button` kann nur wissen, dass sie eine Aktion durch ein Event einleitet. Der Listener des Events, muss eventuell zwischen verschiedenen Quellen des Events unterscheiden und jeweils unterschiedlich reagieren. Da die Quelle des Events in mehreren Fällen ein `Button` sein kann, wird ein Mechanismus benötigt, durch den zwischen zwei `ActionEvent`s unterschieden werden kann. Die Information hierfür muss aus der Schnittstelle des Listeners des Events kommen, da die Quelle wie zum Beispiel ein `Button`, keine Kenntnis von ihren möglichen Einsatzgebieten haben sollte (Komponentenparadigma). Im Falle von Java wäre eine Möglichkeit die Verwendung von `ActionCommand`s. Es wird stattdessen, wie dies etwa auch bei integrierten Entwicklungsumgebungen[31] vorgeschlagen wird, eine *Adapterklasse* generiert, die das `ActionEvent` aufnimmt und eine Methode beim vorgesehenen Listener aufruft, die die eigentliche Aktion tätigt. Dieses Vorgehen wäre auch bei anderen Ereignissen denkbar und ist exemplarisch für `ActionEvent`s realisiert worden. Wenn eine Komponente mit mehreren `Button`s verknüpft wird, muss auch sichergestellt sein, dass die vorgeschlagene Beschriftung und die Verknüpfung mit einer Methode zusammenpassen. Dies wird durch die Programmierung gewährleistet. Ein `Button`, der die Beschriftuung »+« erhält, sollte auch die Addition aufrufen.[32]

[31] Ein Beispiel wäre etwa Borlands JBuilder oder Suns prototypische BeanBox.

[32] Das Resultat der Domänenanalyse könnte neben einem Domänenlexikon (Vokabular, vgl. Abschnitt 5.3) auch Symbole, Piktogramme und

6.3.6.4 Post-Konfiguration

Eine Komponente kann angeben, welche Eigenschaften (Properties) sie für die Post-Konfiguration anbietet. Es wird dann ein Konfigurationstool in Form eines Property-Editors [Java Beans, 1997] für jede Komponente angeboten, mit dessen Hilfe der Anwender gewünschte Einstellungen vornehmen kann. Der Anwender erhält gemäß der Komponentensicht auch eine lauffähige Anwendung, wenn er die Standardvorgaben übernimmt und selbst keine Einstellungen vornimmt.

Es können bei der Post-Konfiguration nur Attribute eingestellt werden, die die Funktionalität der Komponente nicht beeinflussen, sondern bestenfalls ihre Funktionsweise betreffen (zur Unterscheidung vgl. a. [Griffel, 1998]). Es handelt sich also nur um ein zusätzliches *Customizing* der Anwendung. Die Attribute, die zum Customizing freigegeben sind, können den Property-Deskriptoren der Komponenten entnommen werden (s.a. Abschnitt 5.9.1). Über diese kann eine Komponente Standardvorgaben für die eigenen Properties anbieten. Es können ebenfalls Vorschläge für Properties von in der Hierarchie direkt darunter liegenden Komponenten gemacht werden (gegenseitiger Einfluss, vgl. a. Regelsteuerung in Abschnitt 3.10). Wenn Vorschläge von der darüber liegenden Komponente gemacht werden, die Properties mit den gleichen Namen betreffen, dann haben diese Vorschläge Priorität. Typische Einstellungen betreffen zum Beispiel die Vorder- und Hintergrundfarbe, sowie Beschriftungen inklusive Schriftgrösse und verwendeter Schriftart, könnten aber auch zum Beispiel die Art eines Sortieralgorithmus in einer Komponente zur Sortierung betreffen.

6.3.6.5 GUISlots

Bei einer automatischen Anwendungserstellung tritt im Falle zugehöriger grafischer Bedieneroberflächen das Problem des *Layouts*, d.h. einer *sichtbaren* Anordnung der entsprechenden GUI-Komponenten auf. In fast jeder Applikation treten heute Komponenten auf, die in der GUI der Anwendung einen visuellen Anteil besitzen. Daher muss eine der Komponenten der Anwendung eine Art »Gehäuse« für andere Komponenten sein und die Möglichkeit bieten, andere Komponenten anzeigen zu lassen.[33] Die An-

Ähnliches standardisieren, um semantische (Bedien-) Fehler noch weitergehender zu vermeiden.

[33] In [Griffel, 1996] wurden solche »Gehäuse« in Form von Schablonen (Templates) bereitgestellt. Dies wäre auch hier ein denkbarer Ansatz.

zeige erfolgt über die GUISlots. Es ist derzeit nur vorgesehen, eine einzige Komponente mit GUISlots pro Anwendung zu berücksichtigen.[34] Die Auswahl der GUISlots erfolgt über den Komponententyp, da sie Bestandteil der zugehörigen Deskriptoren sind. Die Komponente, die die Slots anbietet, kann eine eigene Methode für die grafische Darstellung eines jeden Komponententyps zur Verfügung stellen. Im der prototypischen Implementation erfolgt technisch gesehen beim Methodenaufruf ein Casting auf eine Java-Klasse, deren Name ebenso wie der Methodenname aus der Schnittstellenbeschreibung der Komponente stammt, die die GUISlots anbietet. Auf ähnliche Weise wäre es zum Beispiel auch möglich, zusätzlich eine Menüleiste zu generieren. Der »Spezialfall« grafischer Komponenten wurde im Rahmen dieser Arbeit nicht allgemein betrachtet, sondern nur spezfisch für Java-Komponenten untersucht bzw. eine Lösung erarbeitet. Das Grundprinzip der vorgeschlagenen Vorgehensweise mit der zugehörigen Erweiterung der Komponenten-Deskriptoren (vgl. Abschnitt 6.2.1.2) eignet sich jedoch als Ausgangsbasis für einen allgemeinen Lösungsvorschlag.

6.3.6.6 Kompilierung und Archivierung

Nachdem der AppBuilder den Quellcode auf Basis des XML-Dokumentes generiert hat, muss der Code noch kompiliert werden. Für die Kompilierung des Quellcodes wird auf die jeweils sprachspezifischen Werkzeuge zurückgegriffen, im Falle der prototypischen Implementation also auf die Java-Runtime. Diese ermöglicht es, auf die Umgebung, in der die Java-Anwendung läuft, zuzugreifen. Es ist mit der Methode exec() möglich den Java-Compiler javac in einem eigenen Prozess aufzurufen und zu verwenden.

Der AppBuilder erstellt eine Liste mit allen Dateien, die für das Anwendungsarchiv (JAR-File) benötigt werden. Die benötigten Dateien sind die Komponenten selbst sowie alle Hilfsklassen und andere Dateien die für die Ausführung der Komponenten benötigt werden. Es werden dafür die Deskriptoren ausgewertet, in denen eine Komponente alle abhängigen Dateien spezifizie-

Dann würde die Domänenimplementierung entsprechende Komponenten bereitstellen. In dieser Arbeit sollte allerdings ein einfacher Mechanismus bereitgestellt werden, der notfalls eine vollständig automatische Erstellung erlaubt.

[34] Dies entspricht der üblichen Sicht eines »Main-Window« in der GUI-Programmierung und sollte daher in den meisten Fällen ausreichend sein.

ren kann. Es wird ein Inhaltsverzeichnis für das Archiv erstellt (in Form einer Manifest-Datei [Java Beans, 1997] und mit allen anderen Dateien in einer JAR-Datei abgelegt. Das Manifest gibt u.a. die Hauptklasse des Jar-Files an, sodass das gesamte Archiv in den meisten Betriebssystemumgebungen zu einer ausführbaren Einheit wird. Für jede Datei muss ein Objekt der Java-Klasse `JarFileEntry` instanziiert und mit `putNextEntry` hinzugefügt werden. Zusätzlich muss jede Datei schrittweise ausgelesen werden und in das JAR-File geschrieben werden.

6.3.7 Erweiterung der Komponentendeskriptoren

Die bereits im Rahmen des Typmanagements eingeführten und als Erweiterung der Beschreibungen des Corba Component Model (CCM) aufzufassenden Deskriptoren werden gemäß den Anforderungen der vorgestellten generativen Verarbeitung nochmals geringfügig erweitert. Ausgangspunkt der Verarbeitung durch den Generator stellt die »Softwarepackage«-Definition, die das Archiv beschreibt, in dem zum Beispiel verschiedene Versionen für unterschiedliche Betriebssysteme vorhanden sein können (vgl. Abschnitt 6.2.1.2) dar. Die Definition des Softwarepackages (`softpkg`, [OMG, 1999]) wurde um die Tags `<gencomponent>`, `feature` und `<guislot>` erweitert.

An Hand eines Beispiels werden im Folgenden die einzelnen Elemente des resultierenden Deskriptors erklärt. Das Beispiel beschreibt eine Komponente vom Typ `Funktionseinheit`, die einen Zwischenspeicher für den Taschenrechner implementiert. Der erste Teil der Schnittstelle ist CCM-konform und stellt eine allgemeine Beschreibung der Komponente dar.

```
<?xml version="1.0"?>
<!DOCTYPE softpkg SYSTEM "softpkg.dtd">
<softpkg name="Memory">
<implementation>
  <code type="Java">
    <fileinarchive name="Memory.class"/>
  </code>
</implementation>
```

Die Komponente ist in Java geschrieben (`<code type=»Java« >`) und befindet sich in der Datei `Memory.class` (`<fileinarchive name=«Memory.class«/>`). Die Spezifikation in diesem Teil kann noch viel detailierter sein, zum Beispiel Angaben über den Autor der Komponente enthalten. Es folgt die vom Generator benötigte Zuordnung eines Merkmalsnamens zu

einem Komponententyp. Das `ferature`-Tag kann dabei entweder direkt Name-Wert-Paare enthalten oder auf einen entsprechenden Feature-Deskriptor verweisen (vgl. a. Abschnitt I.4). Um die Abwärtskompatibilität zu den Standarddeskriptoren bzw. diese verarbeitenden Werkzeugen zu bewahren, sind solche Generator-spezifischen Erweiterungen in das `<gencomponent>`-Tag eingebettet.

```
<descriptor>
  <gencomponent>
    <classname name="Memory"/>
      <componenttype type="funktionseinheit"/>
        <description>
          bietet einen Zwischenspeicher an
        </description>
        <feature name="funktionsart" value="speicher"/>
  </gencomponent>
```

Es kann auf beliebig viele Merkmale mit Hilfe des Feature-Tags verwiesen werden. Der Generator zeigt für diese Komponente später das Merkmal `funktionsart` mit dem Wert `speicher` an.[35]

Eine der verwendeten Komponenten sollte `GUISlot`s für andere Komponenten anbieten. Die Spezifikation hierfür erfolgt mit folgendem Element:

```
<!ELEMENT guislot EMPTY> <!ATTLIST guislot
     methodname CDATA #REQUIRED
     componenttype CDATA #REQUIRED
     guiclass CDATA #REQUIRED>
```

Es wird angegeben, mit welcher Methode eine Komponente eines bestimmten Types angezeigt werden kann. Die `guiclass` ist notwendig, um die anzuzeigende Komponente auf die entsprechende Java-Klasse, die als Argument für die Methode benötigt wird, bei der Erstellung des Anwendungsgerüstes abzubilden (»type cast«).

Insgesamt stehen dem Generator somit durch Auslesen der jeweiligen Komponentenbeschreibungen alle Informationen zur Verfügung, um eine vollständige Anwendung zu konstruieren,

[35] Diese Informationen werden für die Bereitstellung der interaktiven Auswahl der Komponenten während der Spezifikationsphase der Anwendungskonstruktion benötigt. Für einen vollständig automatischen Prozess wäre der ohnehin vom Typmanagement ermöglichte (strukturelle) Vergleich der Merkmalsstrukturen für eine Zuordnung ausreichend.

d.h. die gegenseitigen Verknüpfungen von konkreten Komponenten können automatisch erstellt werden, sobald angegeben wurde, *welche* Komponenten überhaupt vorkommen sollen. Dies geschieht über die eingelesene Architekturspezifikation für eine ausgewählte Produktlinie.

6.4 Unterstützungswerkzeuge

Nicht nur das Erstellen eines entsprechenden Architekturstils, sondern auch die Manipulation der Deskriptoren und der in Ihnen enthaltenen Informationen legt die Bereitstellung entsprechender Werkzeuge zur Vervollständigung der gesamten Umgebung nahe – insbesondere auch um die Akzeptanz des vorgestellten Ansatzes auf Seiten potenzieller Komponentenentwickler zu erhöhen. Auf drei wesentliche Werkzeuge soll dabei im folgenden noch kurz eingegangen werden.

6.4.1 Der Deskriptor-Editor

Die im Rahmen des vorgestellten Ansatzes verwendeten XML-Deskriptoren als (Code-externe) Komponentenbeschreibungen sind letztlich einfache ASCII-Dateien, sodass sie mit jeder Textverarbeitung (bzw. jedem »Editor«) erstellt und modifiziert werden können. Da eine XML-Datei jedoch neben der reinen textuellen Information durch die eingesetzten *Tags* zusätzliche bedeutungstragende Informationen bietet, sollten eher spezifische XML-Editoren eingesetzt werden, die beispielsweise eine entsprechende (farbliche) Syntaxhervorhebung und -überprüfung erlauben.

Auf Grund des Erfolges (d.h. der Akzeptanz) von XML stehen solche Werkzeuge heute zahlreich zur Verfügung. Die für eine Komponentenbeschreibung eingesetzten XML-Dateien tragen allerdings sehr spezifische und zum Teil untereinander abhängige bzw. verknüpfte Informationen, sodass es sinnvoll erschien, dem Komponentenentwickler im Umgang mit diesen Metainformationen die Arbeit zu erleichtern (bzw. dabei möglichen Fehlern vorzubeugen), indem ein spezifisches Werkzeug bereitgestellt wird, das von der Datei-Ebene abstrahiert und Konsistenzprüfungen schon während der Eingabe vornimmt – der Deskriptor-Editor (DE).

Neben der Erstellung und Modifikation der XML-Deskriptoren etwa im Rahmen gemäß Kapitel 4 zu klassifizierender Komponenten, erlaubt der DE auch die Einstellung

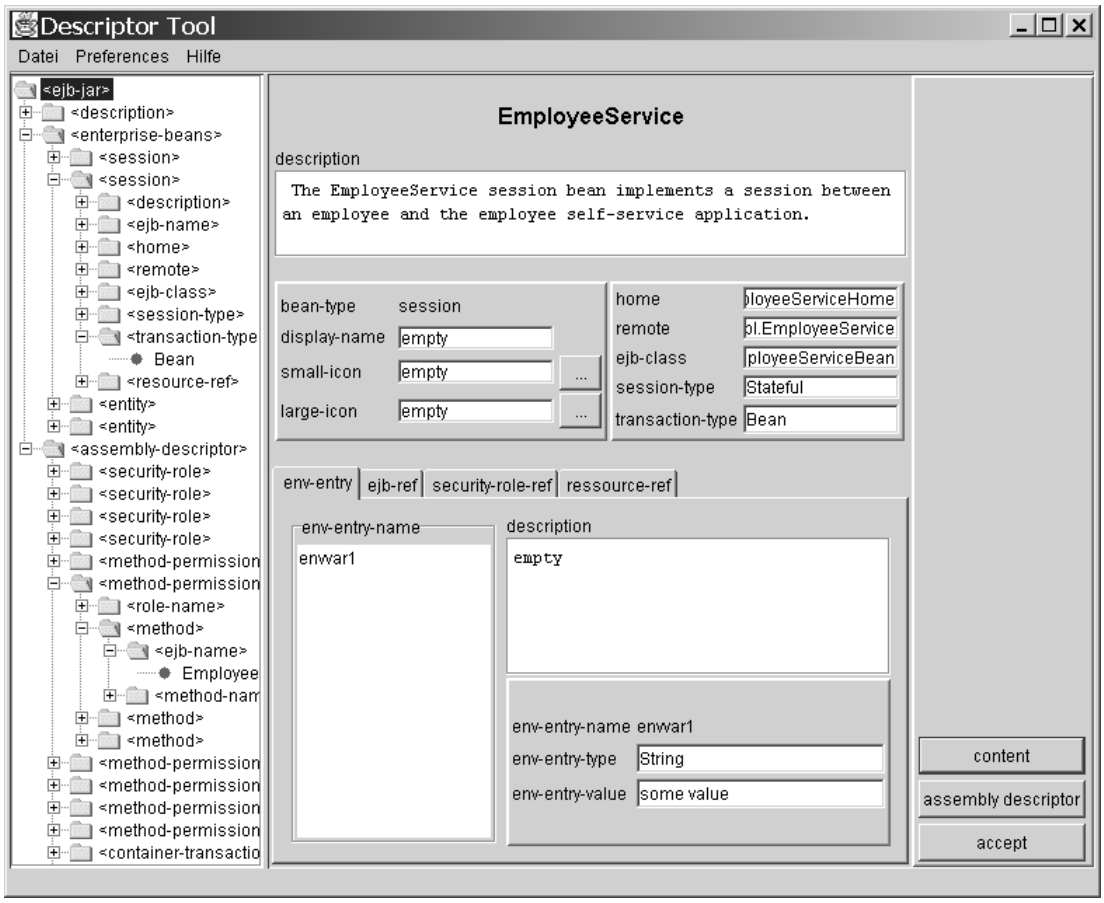


Abbildung 6.17
Hauptansicht des Deskriptor-Editors im EJB-Modus

(Konfiguration) von konkret im Rahmen einer generierten Applikation eingesetzten Komponenten im Sinne einer deklarativen Programmierung Microsoft bezeichnet dies im Rahmen der ».NET«-Initiative als *attributbasierte Programmierung*, d.h. gewünschte Applikationseigenschaften wie etwa eine transaktionale Absicherung werden durch Setzen eines entsprechenden Attributs *aktiviert* und nicht mehr durch explizit zu erstellenden Code ausprogrammiert.

Die notwendigen Funktionalitäten sind dann im Sinne des Komponentenparadigmas bereits vorgefertigt – in der Regel durch sogenannte Container oder über sie vermittelte Dienste bereitgestellt. Hieraus resultiert dann das aktuell grosse Aufmerksamkeit erregende Konzept von *Applikationsservern*, die im Idealfall rekonfigurierbar und weitgehend von ihrer Systemumgebung unabhängig serverseitige Funktionalitäten in Form modu-

lar organisierter (komponentenbasierter) Applikationen bereitstellen. Da die im Rahmen der Tests der Anwendbarkeit der Ansätze der vorliegenden Arbeit erstellten Beispielkomponenten auch entsprechend serverseitige *Enterprise JavaBeans* umfassten, sollte der Deskriptor-Editor insbesondere auch in der Lage sein, die für solche Applikationsserver-Komponenten notwendige Vorbereitung der Einsatzkonfiguration (engl. *Deployment*) durchzuführen (Abbildung 6.17 und 6.18 zeigen ein Standardbeispiel für diesen Fall). Zugleich sollte das Werkzeug entsprechend technologiespezifischen Code aus dem generischen Entwurf des eigentlichen Generators auslagern, da dieser zwar in die Lage versetzt werden sollte, nach der Konstruktionsphase auch ein automatisches Deployment durchzuführen, die entsprechenden Funktionalitäten aber nicht ohne weiteres über verschiedene Komponententechnologien hinweg verallgemeinert werden können. Die Implementation des ursprünglichen Prototypen des Werkzeugs im Rahmen dieser Arbeit sowie generelle Aspekte eines entsprechenden Konfigurationsmanagements sind in [Broschei, 2000] beschrieben.

6.4.2 Der Protokoll-Editor

Der Umgang mit der in Kapitel 3 vorgestellten Syntax zur Beschreibung der Komponenten- bzw. Adapterprotokolle soll dem Komponentenentwickler durch ein entsprechendes Werkzeug erleichtert werden – den Protokoll-Editor (PE). Dieser dient neben der Erfassung entsprechender Schnittstellenspezifikationen insbesondere auch deren Kompatibilitätprüfung, sodass entsprechende Tests während der Entwicklungsphase einzelner Komponenten möglich sind. Dabei sind zwei Einsatzvarianten denkbar:

Der Entwickler spezifiziert zunächst das gewünschte Verhalten als Protokolle in Form entsprechend auszutauschender Nachrichten (Methoden), die dann vom Editor als Quellcodegerüst einer Zielsprache (hier Java) zur Implementation der konkreten Komponenten bereitgestellt werden. Die bevorzugte und ebenfalls vom PE unterstützte Ausgabe sind allerdings die (technologieneutraleren) XML-Deskriptoren.

Die zweite Einsatzmöglichkeit stellt den umgekehrten Prozess dar. Der Entwickler hat eine konkrete Komponente bereits codiert, soll jedoch von der zusätzlichen Aufgabe der manuellen Erstellung entsprechend zugehöriger Protokollbeschreibungen entlastet werden. Hierzu erlaubt der PE das Einlesen der Komponenten, wertet deren entsprechende Methoden aus und

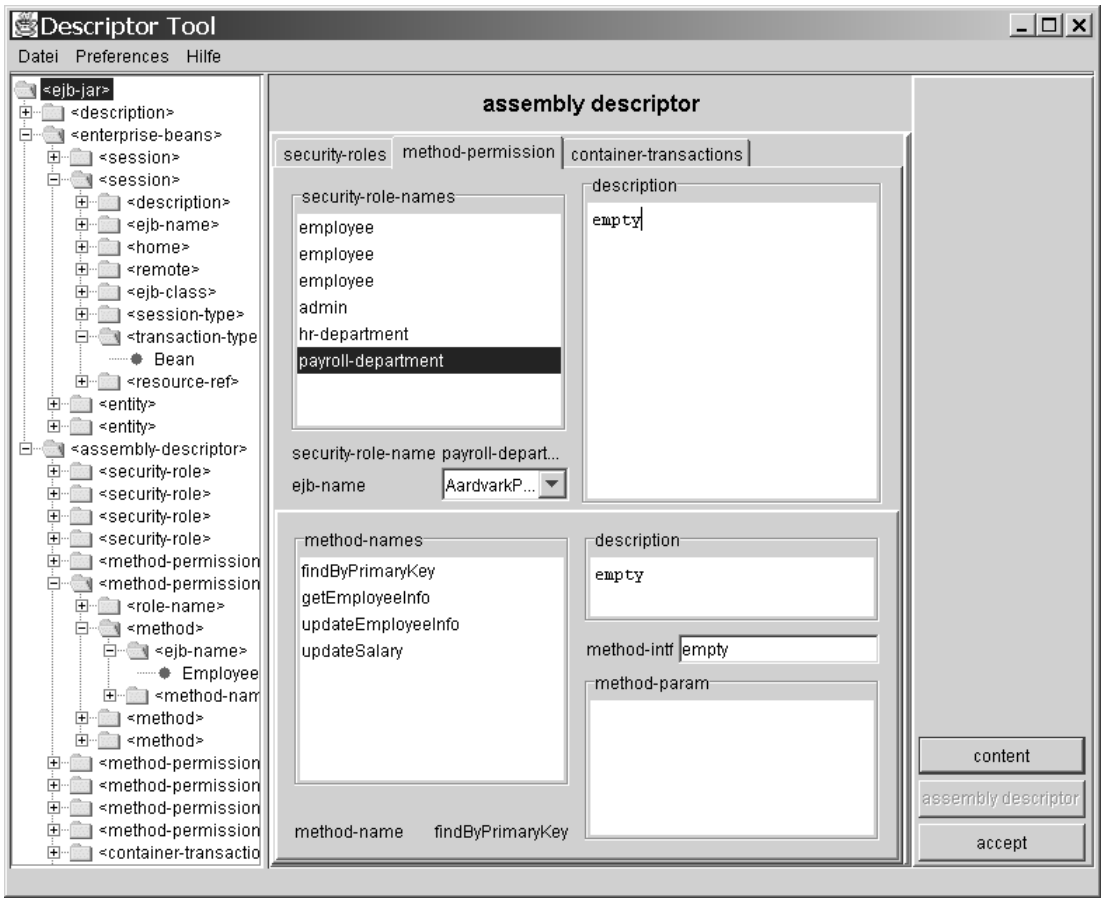


Abbildung 6.18
Bearbeitung eines Assembly-Deskriptors im EJB-Modus

stellt diese zur weiteren Bearbeitung durch den Entwickler in der interaktiven Oberfläche dar.

Der Entwickler kann dann die benötigten Metainformationen wie etwa Richtungsangaben oder Causality-Rules (vgl. a. Abschnitt 3.9.2) ergänzen. Dies muss zwar weitgehend manuell erfolgen, allerdings erleichtert der Protokoll-Editor die Arbeit durch entsprechende Konsistenzchecks und die Übernahmemöglichkeit von Namen und Parametern aus den jeweils dargestellten Informationen.

Für den Fall dass eine Komponente erst entwickelt werden soll, bietet der PE dem Entwickler auch die Möglichkeit ein zu einer eingelesenen Komponente *inverses* Protokoll automatisch zu generieren. Der Entwickler hat damit eine automatisch kompatible Ausgangsbasis zur Verhaltensbeschreibung einer zweiten Komponente, die als Kopplungspartner vorgesehen ist. Das

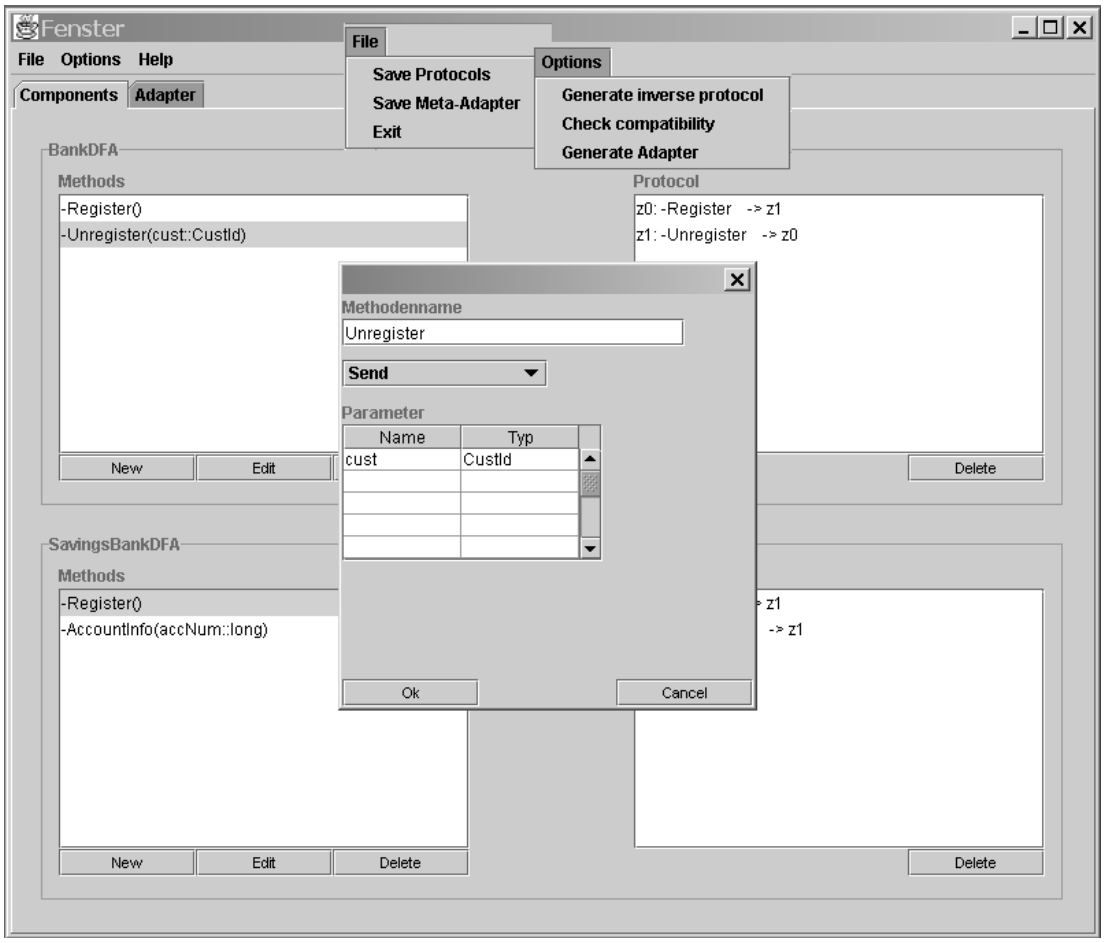


Abbildung 6.19
Hauptansicht des Protokoll-Editors und seiner Menüs

inverse Protokoll kann selbstverständlich dann noch modifiziert und der tatsächlich gewünschten Komponente entsprechend angepasst werden, sodass im Anschluss die Kompatibilität über eine entsprechende Adapterspezifikation wieder hergestellt werden muss.

Der PE erlaubt die vollständige Generierung eines konkreten Adapters im Sinne des Kapitel 3. Die dazu notwendigen Metainformationen erhält er im Falle existierender Komponenten aus der Auswertung des entsprechenden Assoziationsgeflechts der Wissenbasis des Typmanagers[36] sowie den Vorgaben des Entwicklers. Andernfalls erfolgt eine rein manuelle Eingabe (zumin-

[36] konkreter: dem Ausschnitt daraus, der den beiden jeweils gerade betrachteten Komponenten entspricht

dest erleichtert durch die vom PE bereitgestellte Oberfläche). Der erstellte Adapter ist selbstverständlich technologiespezifisch (hier wiederum Java) und kann im Anschluss in der Adapterdomäne des Typmanagers registriert werden, sodass er zur Anwendungskonstruktion bereitsteht. Im Falle des PE-Prototypen wird der Adapter als fertig konfigurierte Komponente in Form eines serialisierten (persistierten) Java-Objekts abgelegt.[37]

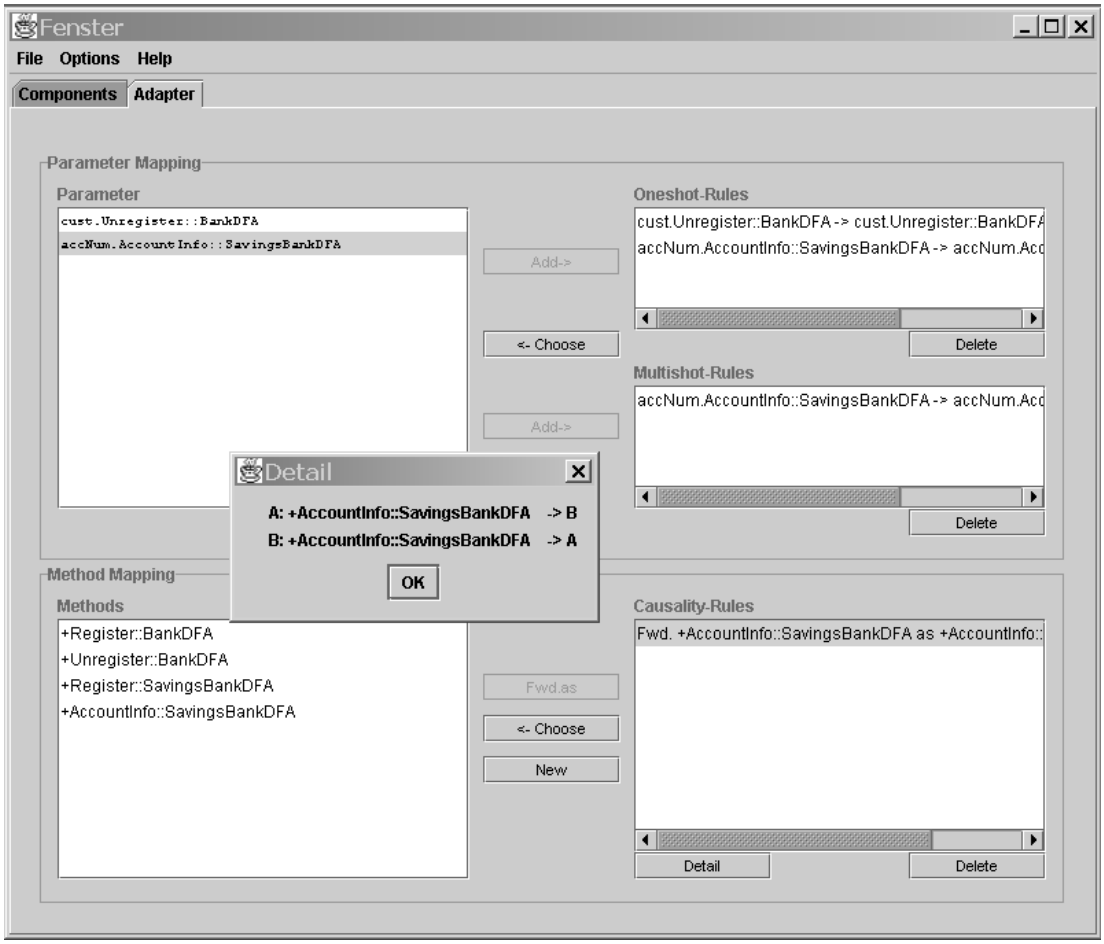


Abbildung 6.20
Vorbereitung der Adapterinformationen im Protokoll-Editor

6.4.3 Der Merkmals-Editor

Ausgangsbasis des dieser Arbeit zu Grunde liegenden Entwicklungsprozesses ist die Idee einer fachnahen bzw. problembezoge-

[37] Dieses kann später auch zur weiteren Verarbeitung / Modifikation des adaptierenden Verhaltens wieder vom PE eingelesen werden.

nen Modellierung einer Anwendungsarchitektur. Diese soll dabei durch Assoziationen von in der jeweiligen Domäne aufgefundenen Fachkonzepten bzw. deren Merkmalen erstellt werden können. Dabei soll das Fachwissen des Architekten im Sinne von Prioritäten und Randbedingungen einfließen können, wobei zugleich ein in der Entwicklungsumgebung vorhandenes Wissen im Sinne einer Typ- und Richtlinienprüfung automatisch wirksam werden und so Fehlentscheidungen frühzeitig minimieren soll.

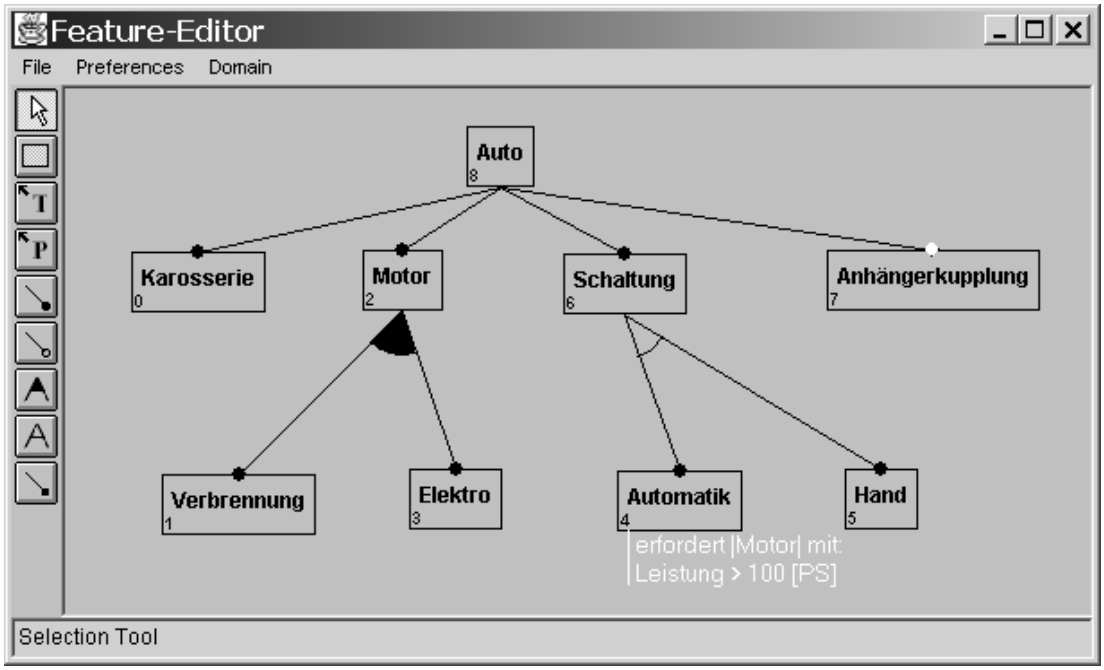


Abbildung 6.21
Ansicht des Merkmalseditors

Als geeignete fachnahe grafische Darstellungsform wurden die Merkmalsdiagramme der FODA-Methode aus Ausgangsbasis gewählt. Entsprechend sollte ein Werkzeug zu deren Erstellung bzw. Manipulation bereitgestellt werden – der Feature-Editor (FE). Da die Implementation des FE im wesentlichen Erich Gammas »JHotDraw«-Framework[38] zur Realisierung grafischer Notationen und damit bekannte Standardtechnologien ausnutzt, soll in diesem Abschnitt nicht auf Implementationsdetails eingegangen werden.[39] Stattdessen soll der Raum für eine kurze Motivation bzw. Diskussion der eigentlichen Darstellungsform genutzt

[38] `http://www.jhotdraw.org`

[39] Die Eigenbeiträge der vorliegenden Arbeit umfassen dabei im wesentlichen Algorithmen zur variablen Darstellung n-fach unterteilter Kreissegmente sowie die Erkennung deren Überdeckungen (vgl. a.

werden. Eine ausführliche Diskussion von Merkmalsdiagrammen findet sich in [Czarnecki & Eisenecker, 2000] und soll hier ebenfalls nicht wiederholt werden. Allerdings wurden einige Darstellungsformen ergänzt, sodass zunächst der Merkmals-Editor aus Benutzersicht betrachtet werden soll.

Abbildung 6.21 zeigt zunächst ein [Czarnecki & Eisenecker, 2000] entliehenes Beispiel für ein Merkmalsdiagramm. Rechteckige Kästen kennzeichnen dabei Merkmale oder Fachkonzepte einer Domäne – hier des Automobilbaus. Das oberste Element einer solchen Hierarchie (eines einzelnen Diagramms) kennzeichnet dabei stets ein Fachkonzept, tiefergelegene Elemente kennzeichnen dann dessen Merkmale, können dabei aber wiederum selbst Fachkonzepte repräsentieren, die in weiteren Diagrammen dann wiederum in ihren Merkmalen dargestellt werden. Merkmale können *obligatorisch* (ausgefüllter Kreis) oder *optional* (leerer Kreis) sein und *alternativ* (leeres Kreissegment) oder in einer kombinatorischen *oder*-Beziehung (ausgefülltes Kreissegment) auftreten. Die Button-Leiste am linken Rand des Editors erlaubt die Auswahl entsprechender Zeichnungselemente. Darüber hinaus können Prioritäten, d.h. die nach Einschätzung des Architekten gewichtete Bedeutung eines Merkmals für die Problemstellung, angegeben werden (in der Abbildung als kleine Zahlen erkennbar). Schließlich können Annotationen im Sinne von Konfigurationsregeln angegeben werden. Abbildung 6.22 zeigt weitere Darstellungselemente des Editors. So können verschiedene Darstellungsformen der Diagramme gewählt werden, um den unterschiedlichen Repräsentationen, die sich in der Literatur finden, gerecht zu werden. Insbesondere kann etwa das Verhältnis der Merkmale untereinander bzw. im Fall eines atomaren Merkmals dessen zugeordneter Basistyp angezeigt werden. Kleine Quadrate übernehmen die gleichen Rollen wie die oben erläuterten Kreise, weisen jedoch darauf hin, dass das entsprechende Merkmal *mehrfach* auftritt.

Da schon während der Anfertigung der Diagramme der FE sowohl auf die Auswertungen des Component-Policy-Managers als auch des Typmanagers zurückgreift, sind die einzelnen Darstellungselemente in Beziehung zu den entsprechenen Typmodellen zu setzen. So handelt es sich bei den Annotationen nicht nur um eine textuelle Informationsangabe für den Betrachter, sondern es können direkt Ausdrücke des hinterlegten Policy-

Abb. 6.21), deren Darstellung hier jedoch den Rahmen sprengen würde, zumal sie keinen inhaltlichen Beitrag zur eigentlichen Thematik leistet.

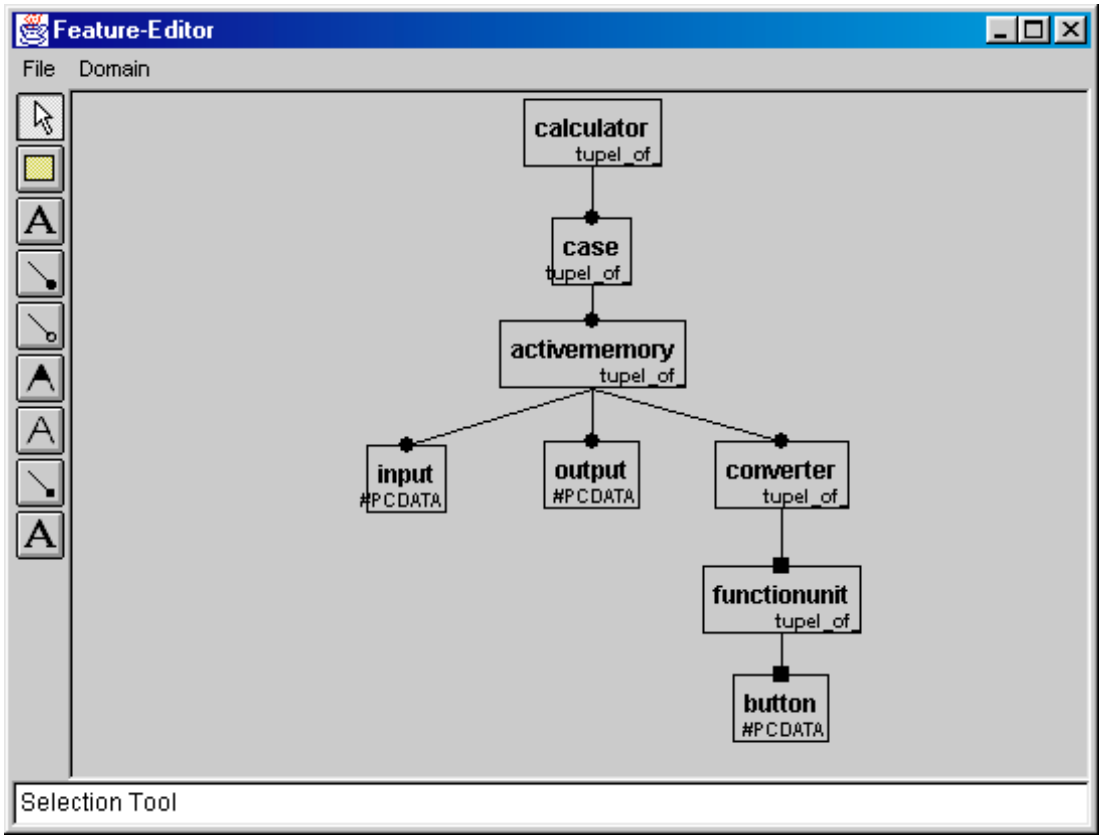
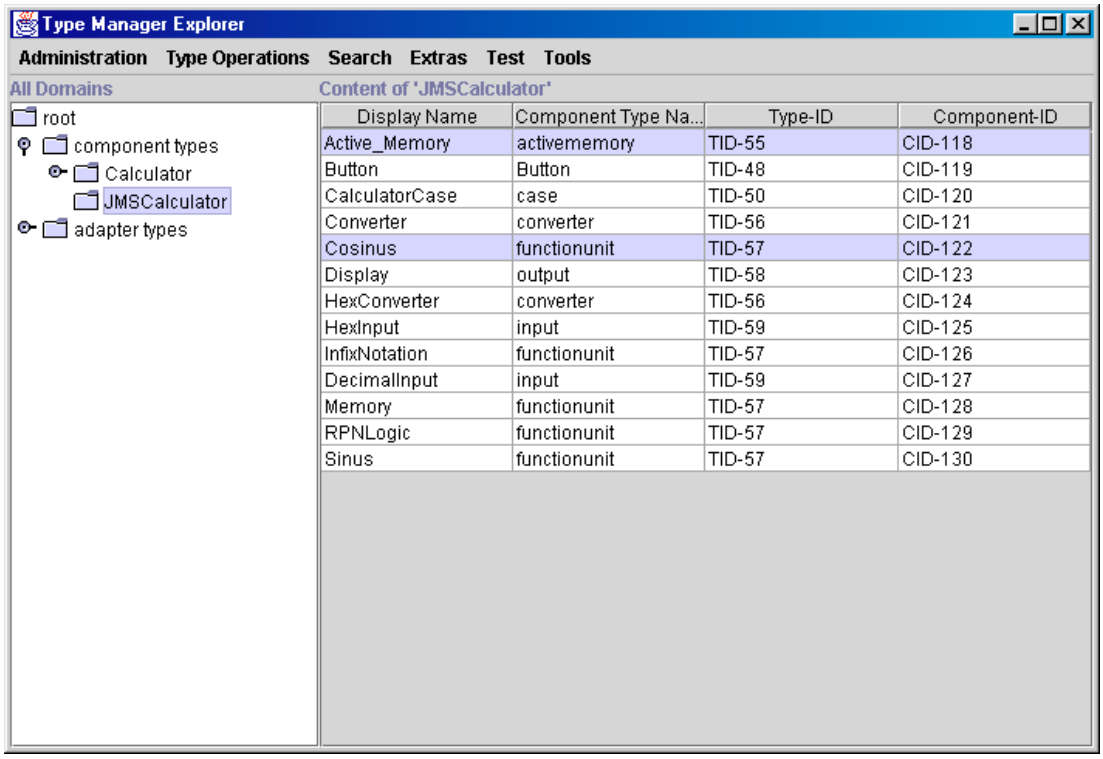


Abbildung 6.22
Weitere Darstellungen des Merkmalseditors

managements verwendet werden (vgl. Abschnitt 3.10). Einzelne Fachkonzepte werden durch Zuordnung eines entsprechenden Namens direkt mit registrierten Komponenten identifiziert (vgl. Abbildung 6.23).

Der Merkmals-Editor erfragt zu diesem Zweck zum Startzeitpunkt die zu betrachtende Domäne vom Architekten und lässt sich im Anschluß vom Typmanager entsprechend vorhandene Komponententypen liefern. Diese stehen dann dem Architekten in der entsprechen Palette des FE bereit, sodass er einerseits von ihm festgehaltene Merkmale einzelnen Komponenten zuordnen, andererseits Merkmale der vorhandenen verwenden kann. Der Deskriptor-zentrierte Ansatz zur Komponentenklassifikation bietet damit zugleich auch eine mögliche Antwort auf die Frage nach dem Verhältnis bzw. dem Zusammenhang zwischen vorgefertigten Komponenten und Anwendungsarchitekturen [Oberndorf et al., 1997].

Eine Ableitung der einzusetzenden Komponenten nur auf Grund der (nicht zugeordneten) Merkmale ist unsinnig, da nur die Komponenten als ganzes über ihren Typ eine aussagekräftige Semantik besitzen. Ein Vermischen von Merkmalen über Komponentengrenzen hinweg erlaubt keine Aussage über die konsistente Verwendung der Merkmale oder des durch sie implizierten Verhaltens. Merkmale sollten damit nur aus Anwendungssicht als solche betrachtet werden, architekturell sind sie bereis konkreten Komponenten zugeordnet. Als dritte unterstützte Variante kann ein dargestelltes Fachkonzept direkt als Komponente identifiziert werden, die dann *alle* Merkmale des entsprechenden Konzepts trägt (vgl. a. Abschnitt 5.10.2).



Display Name	Component Type Na...	Type-ID	Component-ID
Active_Memory	activememory	TID-55	CID-118
Button	Button	TID-48	CID-119
CalculatorCase	case	TID-50	CID-120
Converter	converter	TID-56	CID-121
Cosinus	functionunit	TID-57	CID-122
Display	output	TID-58	CID-123
HexConverter	converter	TID-56	CID-124
HexInput	input	TID-59	CID-125
InfixNotation	functionunit	TID-57	CID-126
DecimalInput	input	TID-59	CID-127
Memory	functionunit	TID-57	CID-128
RPNLogic	functionunit	TID-57	CID-129
Sinus	functionunit	TID-57	CID-130

Abbildung 6.23
Beziehung zum Type-Explorer über Typnamen

Motiviert ist letztere Sichtweise durch die Kapselung domänenspezifischer Features durch jeweils genau eine Komponente in GenVoca (vgl. a. Abschnitt 5.7.1). Entsprechend dieser unterschiedlichen Interpretation der dargestellten Merkmale unterscheidet der Merkmals-Editor verschiedene Arbeitsmodi. Insbesondere dient er neben der Architekturspezifikation zugleich als Werkzeug zur Erstellung der zur Beschreibung von Kompo-

nenten benötigten Merkmalsstrukturen, die dann in das Typmanagement einfließen. Wird ein Diagramm jedoch als Architektur verstanden, kann es als entsprechende DTD gespeichert dem Generator übergeben werden.[40]

Der Generator arbeitet dann ebenfalls in zwei Modi, die nach [Czarnecki & Eisenecker, 2000] als *kompositorisch* bzw. *transformativ* bezeichnet werden. Während ersterer Modus eine vertikale Verfeinerung darstellt und Merkmale im Diagramm als Komponenten interpretiert, stellt der zweite eine horizontale Verfeinerung (bzw. Refaktorisierung) dar und ordnet *mehrere* Merkmale eines Diagramms *einer* Komponente zu. Dies bietet auch einen Ansatzpunkt zur Optimierung der entstehenden Applikation, die allerdings im Rahmen der vorliegenden Arbeit nicht näher untersucht wurde.

Architektur wird in der Komponentensicht häufig als »*Komponenten + Interaktionen*« definiert (vgl. etwa [Oberndorf et al., 1997]), was entsprechend gut den vorgestellten generativen Ansatz unterstützt. Allerdings sind die Interaktionen hierbei implizit (über die zu Grunde liegenden Typbeschreibungen) modelliert und treten nicht explizit im Architekturmodell auf. Dieses betont daher tatsächlich die strukturellen Merkmale und enthält ansonsten nur die Information, dass überhaupt eine Wechselwirkung zwischen strukturell benachbarten[41] Merkmalen vorhanden ist. Die Konfigurationsregeln erlauben dabei die Berücksichtigung möglicher *cross-cut*-Aspekte innerhalb der Gesamtarchitektur.

Es kann also bewusst keine *Funktionalität* grafisch wiedergegeben werden – wie es UML-Diagramme etwa durchaus ermöglichen –, da die Merkmalsdiagramme nur die Struktur widerspiegeln. Die Domänenaktivitäten des Typmodells sind jedoch über die Diagrammkanten impliziert, wenn die Merkmale als fachliche Domänenkonzepte interpretiert werden. Eine eingezeichnete Kante heißt dann, dass *alle* Kompatibilitätsbedingungen zwischen den verbundenen Komponenten erfüllt sein müssen. Der derzeitge Prototyp des Merkmals-Editors unterstützt weder

[40] Daher verschwimmt auch die Grenze einer DSL (domänenspezifischen Systembeschreibungssprache) und einer Komponentenverknüpfungssprache (ICCL) (vgl. Abschnitt 5.7.3.2). Während erstere im vorgestellten Ansatz durch generische XML-Ausdrucksmittel mit der Spezialisierung entsprechender DTDs und Deskriptoren ersetzt wurde, tritt letztere gar nicht explizit in Erscheinung. Stattdessen werden benötigte Informationen direkt aus dem Typmodell sowie den konkreten Komponenten abgeleitet.

[41] vgl. a. die Demeter-Forderung, Abschnitt 5.5.2

die denkbare explizite Kantenbeschriftung mit den Mengen der ausgetauschten Business Events noch die getrennte Berücksichtigung einzelner Komponentensegmente in der Zuordnung der Merkmale. Diese Möglichkeiten sind sicherlich nicht uninteressant, standen jedoch nicht im Vordergrund der Realisierung, da eine Interaktion ohnehin nicht segementübergreifend definiert ist und sich andererseits die Frage nach der semantischen Vollständigkeit eines Fachkonzepts stellen würde, wenn der Architekt nur einzelne Segmente selektieren dürfte. (vgl. a. die Diskussion der signifikanten Merkmale in Abschnitt 5.10.2).

Bei der Auswahlentscheidung für eine grafische Notation wurde bewusst nicht die derzeit so beliebte UML-Notation gewählt, da sie die für den späteren generativen Prozess gewünschte Variabilität[42] durch ihre aggregierenden bzw. Sub- / Supertyp-Konstrukte zu frühzeitig und zu stark einschränkt [Czarnecki & Eisenecker, 2000] und nach Meinung des Autors und anderen für einen fachnahen Modellierungsansatz zu sehr von technischen (objektorientierten) Details geprägt ist. So bezeichnen etwa [Oberndorf et al., 1997] lediglich die »Booch«-Anteile[43] der UML als für eine architekturzentrierte Modellierung geeignet, während die OMT-Anteile (Rumbaugh) nur für technische Details eingesetzt werden sollten. Dagegen passen Jacobsons Use-Case-Diagramme recht gut zu einer komponentenbasierten Modellierung im Sinne dieser Arbeit, da sie Aktoren und deren *Interaktionen* beschreiben (vgl. a. [Griffel, 1998]), ohne technische Details zu beinhalten. Zwar haben sich Autoren wie [Kruchten, 1995] bereits frühzeitig mit der Ausnutzung der UML zur Architekturbeschreibung von Systemen auseinandergesetzt, jedoch werden sich entsprechende Strömungen wohl erst in den nächsten Jahren mit der zunehmenden Abstraktion von der einzelnen Programmzeile und der Überarbeitung der UML durchsetzen können. Auch andere auf den ersten Blick vielversprechende Diagrammarten der UML wie das Objektdiagramm und das Sequenzdiagramm sind leider nicht unproblematisch. So kann ein Objektdiagramm zwar als Exemplar eines Entwurfs-

[42] Sollen gemeinsame Merkmale eines Konzepts ausgedrückt werden führt dies beispielsweise zur Bildung von abgeleiteten Subklassen, die dann im konkreten System zwar verwendet werden können, jedoch die entspechenden Merkmale nur noch gebündelt zur Verfügung stellen. Entsprechend führt diese Vorgehensweise typischerweise auch zur Bildung »künstlich« motivierter Klassen und damit zur unnötigen Zunahme von deren Gesamtzahl.

[43] Die UML ist ja aus der »Vereinigung« der drei Ansätze von Rumbaugh, Booch und Jacobson hervorgegangen [Booch et al., 1998].

musters interpretiert werden, umfasst dann aber wiederum gerade nicht die Variabilität wie sie im Rahmen des hier angestrebten »family engineerings« zur Beschreibung einer ganzen Klasse von Systemen notwendig ist [Allen, 1997]. Sequenzendiagramme sind auf ähnliche Weise »zu konkret«, indem sie ein Interaktionsmuster durch ein konkretes Komponentenverhalten definieren. Der obige Ansatz geht jedoch davon aus, eine Interaktion in der Architekturrepräsentation lediglich zu vermerken (eine Verbindungslinie), das explizite Interaktionsmuster sich jedoch aus den beteiligten Komponenten selbst ergibt – je nach dem wie deren konkrete Ausprägungen es benötigen.

Es bleibt abschließend anzumerken, dass der Ansatz dieser Arbeit nicht versucht Architekturen »herzuleiten«, da automatische Ansätze in dieser Richtung bislang stets gescheitert sind [Kazman & Carriére, 1997]. Stattdessen soll der Entwickler lediglich bei der Erstellung von Architekturspezifikationen unterstützt werden. Entsprechend ableitende Ansätze versuchen Architekturen aus bestehenden Entwicklungsartefakten zu extrahieren. Dies entspräche im vorliegenden Ansatz am ehesten der Analyse der Verknüpfungsmöglichkeiten zwischen Komponenten. Ein entsprechendes Vorgehen wurde hier jedoch nicht untersucht, da eine generative Erstellung und nicht etwa ein analytisches Reengineering von Applikationen im Vordergrund der Betrachtungen stand.

7 Resümee und Ausblick

Die vorliegende Arbeit leistet einen wesentlichen Beitrag zur Schließung der in den gängigen Softwareentwicklungsprozessen bestehenden Lücke zwischen der fachnahen, problembezogenen Modellierung einer Softwareapplikation einerseits und deren konkreten technischen Implementation andererseits.

Diese »*application development gap*« liegt zum einen im nach wie vor mangelnden Problembewusstsein der Entwickler begründet, die eine abstrahierende Denkweise oftmals zu Gunsten der konkreten Codierung opfern und somit *fachnahe* Modelle gar nicht erst entstehen lassen. Zum anderen orientieren sich die heute in der Praxis eingeführten Modellierungskonzepte oftmals selbst zu sehr an technischen Details. In der Folge entstehen dann Einzelanfertigungen von Softwareapplikationen, die eine projektübergreifende Ausnutzung ihrer (Teil-) Resultate im Sinne einer Wiederverwendung vermissen lassen.

Immerhin kann die Informatik zumindest in den letzten Jahren erste Erfolge einer Wieder- bzw. Weiterverwendung von Entwicklungserkenntnissen verbuchen, nämlich im Sinne abstrakter Bauvorschriften in Form von Entwurfsmustern. Dies legt die Hoffnung nahe, dass sich doch langsam ein *architekturzentriertes* Denken in der Entwicklergemeinde etabliert. Das derzeit wichtigste Umsetzungsmittel solcher Muster in Form von Rahmenwerken erweist sich jedoch immer wieder als problematisch, sei es auf Grund komplexer Einarbeitungszeiten, implizit verborgenen Abhängigkeiten oder schlicht des schnellen »Alterns« verwendeter Implementationstechniken.

Hier erscheint ein *komponentenbasierter Ansatz* als mögliche Lösung, da er konkrete Technologie und architekturelles Denken per se beinhaltet, aber besser trennt. Solche Versprechungen können sich aber nur erfüllen, wenn auch der konkrete Umgang mit Komponenten*technologien* ausreichend abstrahiert werden kann und zugleich den Anwendungsentwicklern Werkzeuge an die Hand gegeben werden, um tatsächlich fachnahe Formulierungen ihrer Systemkonzepte vorzunehmen.

Ein Komponentenansatz bietet dann nach Ansicht des Autors ideale Voraussetzungen zur Schaffung eines bruchlosen Übergangs, d.h. der Erhaltung der ursprünglich konzipierten Eigenschaften in der tatsächlichen Implementation, da jede einzelne Komponente bereits eine deutliche Abstraktion ihrer Umsetzung darstellt und entgegen früheren Softwarekonstrukten umfangreiche Informationen über sich und ihre Einsatzmöglichkeiten »mitliefert«. Dies erfüllt sich aber wiederum nur dann, wenn diese *theoretischen* Eigenschaften eines Komponentenparadigmas auch *praktiziert* werden können, d.h. wiederum Mechanismen und Konzepte bereitstehen, die Komponenten unabhängig von ihrer Implementationstechnik entsprechend semantisch reich beschreib- und manipulierbar werden lassen.

Die Idee entsprechender *Schnittstellenbeschreibungen* ist nicht neu und spätestens seit ihrer Verwendung zur Abstraktion entfernt zugreifbarer Dienste in verteilten Systemen allgemein bekannt. Die vorliegende Arbeit schlägt jedoch vor, sich zur Charakterisierung einer Komponente nicht auf solche strukturellen Spezifikationen ihrer »Oberfläche«, sondern auf die ihr möglichen *Interaktionen*, d.h. ihr *beobachtbares Verhalten* zu konzentrieren. Eine solche Interaktionsorientierung bietet den Vorteil einer inhärenten Abstraktion von konkreten Komponentendetails und stellt stattdessen deren *Einsatzmöglichkeiten* in den Vordergrund. Gelingt es nun, Interaktionen mit problembezogenen, bei der Abwicklung einer Aufgabe, die von einer Software unterstützt werden soll, auftretenden Aktivitäten zu assoziieren, ist ein geeignetes Mittel zur Verbindung von Modell und Bausteinen zu dessen Materialisation gefunden. Allerdings muss ein solcher Ansatz dann auch wiederum eine Abbildung des fachnahen Interaktionsbegriffs auf entsprechend technisch realisierbare *Kommunikationsmittel* beinhalten.

Damit Anwendungsentwickler auch tatsächlich während der gesamten Entwicklungszeit in ihrer Rolle als Fachleute für einen Anwendungsbereich arbeiten können, müssen diese technischen Übergänge dann ebenfalls vor ihnen verborgen werden. Der entsprechend einzig gangbare Weg scheint dann in der *automatischen Erstellung* einer Applikation zu liegen, d.h. ihrer *Generierung*. Auch für ein generatives Vorgehen erscheint ein komponentenbasierter Ansatz mit seinen vorgefertigen Bausteinen eine ideale Voraussetzung zu sein, da einerseits wieder umfangreiche Informationen zu den einzelnen Komponenten vorliegen, andererseits die Komplexität ihrer Verknüpfung (sowohl bezogen auf die Anzahl als auch auf den Grad der Wechselwirkung) deutlich

geringer ist, als etwa für objektorientierte oder gar rein auf prozeduralem Quellcode basierende Strukturen.

Entsprechend diesen Überlegungen und der Darstellung der allgemeinen Motivationen in Kapitel 1 hat die vorliegende Arbeit Antworten auf die einzelnen Fragestellungen gegeben und eine zugehörige integrative Gesamtsicht in Form einer vollständigen Entwicklungsumgebung bereitgestellt. Dabei wurden zunächst in Kapitel 2 die grundlegenden Aspekte und Anforderungen an ein Komponentenmodell diskutiert und dabei vor allem die Rolle bzw. das gegenseitige Verhältnis von Struktur und Verhalten und insbesondere deren Beschreibungs- bzw. Spezifikationsmöglichkeiten betrachtet. Ergebnis ist ein azyklisches, streng hierarchisches, segmentiertes, aber aggregierendes Komponentenmodell, das bewusst Einschränkungen in der Ausdrucksmächtigkeit (in Bezug auf denkbare Anwendungsstrukturen) hinnimmt, um einen klareren Komponentenbegriff zu erhalten und die vielfach in anderen Modellen auftretenden konzeptuellen Probleme zu vermeiden, wie etwa der nebenläufigen Mehrfachverwendung oder der semantischen Unvollständigkeit.

Um das Modell formal zu fassen, wurde die strukturelle Ausdrucksmächtigkeit der Feature-Logik mit ihren dynamischen Fähigkeiten zur Typisierung von Merkmalsmengen eingesetzt, um die im Komponentenmodell auftretenden elementaren Konstrukte – wie etwa Schnittstelle und Nachricht – in ihren gegenseitigen Beziehungen zu beschreiben. Dabei wurde zugleich eine Form temporaler Logik vorgeschlagen, die kausale und verhaltensmäßige Beziehungen entsprechender Elemente formal zu fassen vermag. Durch Einbettung temporaler Propositionen als Terme des Feature-Logik-Modells ist ein integriertes Modell zur Spezifikation eines Softwarebausteins – insbesondere auch im Falle eines Aggregats (Kompositum) – entstanden. Dabei wurde jedoch zugleich die Abhängigkeit solcher formalen Ansätze von konkreten *Standardisierungen* der Anwendungsbereiche (Domänen), für die sie eingesetzt werden sollen, deutlich.

Ferner erschien der Verlust an (selbstbeschreibender) Information durch den unifizierenden Charakter der Einzelspezifikationen im Falle einer Komposition unbefriedigend, sodass in Kapitel 3 ein operationales Modell basierend auf der Auswertung endlicher Automaten bereitgestellt wurde, das insbesondere die semi-automatische Ableitung eines *adaptierenden* Verhaltens erlauben sollte. Entsprechende Adapter erscheinen für einen Ansatz, der ausschließlich von vorgefertigen Komponenten ausgeht, unabdingbar, um die Auswahlmöglichkeiten bzw. den Grad der Wiederverwendbarkeit der ansonsten geschlosse-

nen Komponenten zu erhöhen. Das Adaptionskonzept hat dabei allerdings gegenüber einer rein technischen Anpassung die Adaption des wechselseitigen *Verhaltens* zweier Komponenten in den Vordergrund gestellt, da es als Grundlage für den Interaktionsbegriff in der weiteren Arbeit dienen sollte. Es wurden Algorithmen zur Überprüfung einer *Konformität* – also eines gleichartigen Verhaltens – und einer *Kompatibilität* – also eines kollaborierenden Verhaltens – vorgestellt und formal nachgewiesen. Entspechende in der Literatur zu findende Ansätze wurden dabei korrigiert und entsprechend den Anforderungen des Komponentenmodells erweitert. Zentral ist also die Betrachtung der *Vorgänge* im Gegensatz zur Betrachtung der *Aktoren*.

Entsprechend prägt Kapitel 4 den Begriff des *Interaktionsmusters* zur Kennzeichnung von Komponentenschnittstellen und definiert hierüber einen interaktionszentrierten *Typbegriff* für Komponenten, der im Rahmen eines vorgeschlagenen Typmanagements zur Klassifikation und Beurteilung von Softwarebausteinen dient. Das entworfene Typmodell ist dabei *mehrstufig*, um einerseits einen fachnahen und damit Bedeutung tragenden Interaktionsbegriff aufrecht erhalten zu können, andererseits aber eine Abbildung auf konkrete technische Kommunikationsmittel zu erlauben. Zugleich sollte die durchgängige Verwendung eines Nachrichtenkonzepts die Abbildbarkeit auf möglichst viele existierende wie zukünftige konkrete Komponentenmodelle erlauben.

Der Begriff der Semantik wird im Rahmen des Typmodells über strukturelle Vergleiche sogenannter Merkmalsstrukturen definiert, die einerseits die direkte Assoziation mit einer *Domänenterminologie* bzw. den Fachkonzepten eines Anwendungsbereichs erlauben, andererseits mit der zuvor eingeführten Feature-Logik als formalem Verarbeitungsmodell harmonieren. Die Beschreibung eines Komponententyps liegt dann in Form mehrerer sogenannter Komponentendeskriptoren vor, die in ihrer Gesamtheit das Interaktionsverhalten und die entsprechende operationale Schnittstelle einer Komponente sowohl problem- als auch implementationsspezifisch beschreiben und so den gewünschten bruchlosen Übergang erlauben. Dieses interaktionsorientierte Typmodell erlaubt insbesondere einen flexiblen Umgang mit unterschiedlichsten *Typrelationen* auch über die klassischen strengen Konzepte des Sub- / Supertyps hinaus, indem der Begriff der *Affinität* von Komponententypen eingeführt wird. Auf diese Weise kann das Typkonzept bei Bedarf auch um weitere Relationen ergänzt bzw. eine Komponente in ihrem Verhältnis zu anderen praxisnah beurteilt werden.

Das Typmanagement bildet dann auch die Grundlage für einen automatischen Konstruktionsprozess, der in Kapitel 5 vorgestellt wird. Dabei erfolgte zunächst eine Einordnung in bestehende Ansätze einer generativen Sichtweise auf Software, insbesondere die Erkenntnisse verwandter Herangehensweisen wie die adaptive oder aspektorientierte Programmierung sollten in den Entwurf einfließen bzw. wo nötig abgegrenzt werden. Ein derart vollständig auf konkreten Bausteinen aufsetzender generativer Ansatz wie der hier vorgestellte ist dabei bislang in der Literatur nicht aufgetreten, die sich in der Regel klassisch auf der Programmzeile des Quellcodes bewegt, da sie Ziele wie etwa einen optimal kompakten Code verfolgt. Die vorgestellte generative Softwarekonstruktion ist dabei vom Leitbild der Erzeugung ganzer Produktlinien, also einer ganzen Variationsbreite einer Applikation, geprägt. Die Variabilität entsteht dabei zum einen durch die Auswahlmöglichkeit des Generators aus einem Pool von Komponenten zum anderen durch den angeleiteten manuellen Einfluss des Softwarekonstrukteurs auf die *Konfiguration* einer Applikation bzw. ihrer Komponenten.

Um diese Variabilität bzw. den Erfolg automatischer Entscheidungen des Generators nicht zu gefährden, geht der Entwicklungsprozess dabei zunächst vom Entwurf einer domänenspezifischen *Architektur* aus. Die Analyse der Architektur und die daraus folgende Korrelation mit vorhandenen Komponententypen erlaubt die Erstellung einer konkreten *Systemstruktur* durch den Generator und die anschließende Einbettung vorhandener Komponenten in ein solches Anwendungsskelett. Erst bei dieser Konstruktion durch den Generator werden *Instanzeigenschaften* der Komponenten berücksichtigt. Die Einbindung einer Richtlinienverarbeitung dient dabei wo notwendig der Durchsetzung von Typconstraints bei konkreten Instanzen – wenn also nur eine Komponente »Ellipse« vorliegt, ließe sich diese typkonform als »Kreis« einsetzen, wenn das Typconstraint »Radius1 = Radius2« berücksichtigt wird. Dann muss dieses aber auch für die tatsächlich eingesetzte Komponente *gelten*, was durch die *Aktivierung* der (gleichlautenden) Policy während der Verknüpfung geschieht. Die Implementation verschiedener Beispielkomponenten unter Verwendung unterschiedlicher lokaler wie verteilter Implementationstechniken hat dabei gezeigt, dass das vorgeschlagene Generatorkonzept mit dem zu Grunde gelegten interaktionsorientierten Komponentenmodell in der Tat in der Lage ist, weitgehend technologie- und domänenunabhängig zu arbeiten.

Da sich in der Literatur üblicherweise seit Jahren immer wieder theoretische Modelle und Betrachtungen zu auch für die vorliegende Arbeit relevanten Themen finden, ohne dass diese bislang einen nennenswerten Eindruck in der allgemeinen Entwicklungspraxis hinterlassen haben, lag ein wichtiges Ziel dieser Arbeit in der vollständigen *Umsetzung* sämtlicher betrachteter Mechanismen und Konzepte, zum einen als Nachweis ihrer praktischen Tragfähigkeit zum anderen auch als konkrete »Werbemaßnahme« für die vertretenden Ansätze der Fachnähe und Interaktionskonzentration sowie vor allem auch der generativen *Produktion* von Software, um nicht nur das akademische, sondern auch das praktische Bewusstsein zu wecken. Denn die Informatik sollte nach Meinung des Autors an diesem Punkt nicht Selbstzweck sein, sondern sich zu einer konkreten Ingenieursdisziplin entwickeln. Entsprechend gibt Kapitel 6 einen Überblick über wesentliche Teile der dieser Arbeit zu Grunde liegenden Implementationsarbeiten, sowohl ihrer Kernelemente »Typmanager« und »Generator«, aber auch ihrer Hilfsmittel wie adaptierende Konnektoren und Werkzeuge für den Entwickler. Aus Platzgründen und zur Konzentration auf den eigentlichen Konstruktionsprozess wurden dabei nicht alle für die gesamte Entwicklungsumgebung notwendigen Details erörtert, so sind etwa die kompletten Aktualisierungsmechanismen für eine fertig erstellte Applikation außen vor gelassen und es ist nur jeweils auf entsprechende Arbeiten und Veröffentlichungen verwiesen worden. Im Rahmen des Implementationskapitels finden sich auch einige vergleichende und konzeptuelle Betrachtungen, insbesondere im Zuge der konkreten Umsetzung entsprechender Algorithmen sowie allgemeinen Betrachtungen zu den verwendeten Darstellungsformen und Datenstrukturen. Hervorzuheben ist dabei beispielsweise die Möglichkeit eines *konstruktiven Parsings* mittels entsprechender XML-Werkzeuge oder die Vermeidung einer »seitenverkehrten« Einbindung von Adaptern.

Insgesamt umfassen die wesentlichen Beiträge der vorliegenden Arbeit neben einem umfassenden Überblick des aktuellen Standes der Komponentendiskussion insbesondere

- die Etablierung eines Richtlinien-kontrollierten, typsicheren Kompositionsmodells,
- die Konzeption und Realisierung eines expliziten, fachorientierten Typmanagements für Softwarekomponenten,
- die Bereitstellung eines interaktionsorientierten Adaptionskonzepts,

- den Entwurf und die Implementation eines Generators zur konstruktiven Softwaresynthese unter Verwendung der vorangehenden beiden Punkte und schließlich

- die Einbettung der erarbeiteten Konzepte in die Entwicklungspraxis moderner im Markt befindlicher Komponentenansätze durch die Bereitstellung einer vollständigen Entwicklungsumgebung.

Die Arbeit hat dabei unter anderem gezeigt:

- Die Etablierung eines Generator-basierten Ansatzes wird durch ein Komponentenmodell erleichtert; umgekehrt ist dies auch ein Nachweis für die an das Komponentenparadigma gestellten Erwartungen, da *automatische* Prozesse offenbar von ihnen profitieren.

- Die Verwendung eines generativen Komponentenansatzes ermöglicht es, *Individual*software aus *Standard*bausteinen zu erstellen und so beide in der jetzigen Entwicklungspraxis eher konträren Entwicklungsichten zu vereinigen, bzw. deren jeweilige Vorteile unter Vermeidung entsprechender Nachteile bereitzustellen.

- Die Einführung eines semantischen Verständnisses wechselseitiger Aufrufe einzelner Softwareteile ist möglich, sodass sich eine prozessorientierte mit einer strukturellen Systemsicht verbinden läßt.

- Ein generativer Ansatz hat den Vorteil, die Konformität der entstehenden Produktlinie zur Architekturspezifikation zu *implizieren* und entsprechende Tests im *Nachhinein* zu minimieren, d.h. es handelt sich um eine Vorab-Qualitätssicherung wie sie in der Softwarewareentwicklung das aufwändige »Abschlußdebugging« ersetzen bzw. wenigstens begleiten sollte [Griffel, 2000c]. Der Autor bezeichnet dies auch als »*Correctnes-by-Construction*«.

- Die vorgeschlagene Generative Softwarekonstruktion kann als eine Art *just-in-time* Anwendungsentwicklung mit entsprechenden Kosten- und Ablaufoptimierungen verstanden werden.

- Ein neuartiges Hilfskriterium steht zur Identifikation von Komponenten bereit: Ein Softwarebaustein wäre genau dann eine Komponente, wenn eine vollständige funktionale

Abdeckung eines Domänenkonzepts derart erfolgt, dass keine Mehrfachverwendung von Komponenten innerhalb einer Applikation mehr auftritt.

Ausblickend sollten zukünftig noch einige Punkte für das vorgeschlagene Gesamtkonzept betrachtet werden. So schlägt [Allen, 1997] vor, die eingesetzten Merkmalsdiagramme auch zur Darstellung von Interaktionsmustern einzusetzen. In der vorliegenden Arbeit könnten dazu in analoger Interpretation zu den von Allen verwendeten CSP-Operatoren »muss«-Kanten als sequenzielle Zusammenhänge, »exklusiv-oder«-Kanten als Alternativen und eine echte »oder«-Kante als synchronisierendes Miteinander in Form einer Interaktion betrachtet werden. Diesen interessanten Ansatz hat die vorliegende Arbeit jedoch noch nicht verfolgt, eine entsprechende Erweiterung würde aber die Homogenität der Verarbeitung weiter verbessern.

Generatoren führen per Definition automatiserbare Aufgaben aus, im Rahmen der Softwareentwicklung also etwa die Code-Erzeugung, die Komposition, die Kompositionsvalidierung und die Optimierung. Die vorliegende Arbeit hat nur die mittleren beiden Aspekte betrachtet, da für komplexe Systeme Eigenschaften wie die Performanz, Robustheit etc. eher von der Architektur abhängen, als von einzelnen Codierungsarten und Spracheigenschaften [Klein et al., 1998]. Zukünftig wäre eine nähere Untersuchung der Optimierungsmöglichkeiten im Rahmen des vorgeschlagenen Prozesses aber durchaus wünschenswert.

Die gewählte streng hierarchische Form der Verknüpfungen im Rahmen der vorgeschlagenen DTD-Architekturspezifiaktion stellt eine Einschränkung dar, die zu »unnatürlichen« Architekturmodellen führen kann. Einen flexibleren Einsatz und zugleich eine bessere Unterstüzung entsprechender Optimierungsmöglichkeiten versprechen die Möglichkeiten der kommenden XPath-Spezifikationen für XML (XPointer, XLinks) [W3C, 2000b] bzw. die Typisierungsbemühungen für XML im Rahmen von XSchema bzw. der Document Content Definition (DCD) [W3C, 2000a]. Für die Navigation durch die Architekturspezifikation könnte dann eine Adaption des Navigationsalgorithmus aus dem Demeter-Ansatz (s.a. 5.5.2) zum Einsatz kommen.

Wichtig im generativen Ansatz ist im Gegensatz zu den in Kapitel 5 ebenfalls diskutierten Quellcode-basierten Techniken, dass die ausgewählten Komponenten in der Anwendung »erhalten« bleiben und nicht in einem (optimierten) Gesamtcode aufgehen. D.h. aber auch, dass die Möglichkeit besteht, später Einzelkomponenten noch auszutauschen (Produktpflege). Daher

sind aber auch COTS-Überlegungen[1] genereller Art von Bedeutung für den vorgestellten Ansatz, beispielsweise die Auswahl eines Komponentenanbieters, das Auffinden von entsprechender Märkte, Fragen der Lizenzierung, Rechtliche Aspekte der Wiederverwendung, Versionspflege, Amortisierungskosten etc.

Nicht zu vernachlässigen sind auch die psychologischen Hemmnisse einer konsequenten Forderung nach einer Industrialisierung der Softwareentwicklung und der Etablierung einer entsprechenden generativen Sichtweise, da sie zu einer Ablehnung der Konzepte bei den involvierten Akteuren führen kann. Diese werden praktisch in zwei Klassen geteilt, wobei die einen in vermeintlich kreativerer Weise Komponenten implementieren, während die anderen auf deren bloße Rekombination beschränkt sind. Hinzu kommt das bekannte *NIH-Syndrom* (not invented here), das für das Mißtrauen gegenüber nicht selber erzeugter Software steht. Um das volle Potenzial von Komponentenansätzen nutzen zu können, scheint daher die Motivation eines neuen pragmatischeren Verständnisses der Softwareentwicklung von der Kunstform hin zu einem Produktionsprozess Voraussetzung zu sein.

Abschließend bleibt anzumerken, dass die nach Fertigstellung der vorliegenden Darstellung vorgenomme vollständige Überarbeitung und Integration der ursprünglichen Prototypen eine robuste Entwicklungsumgebung zur Unterstützung einer Generativen Softwarekonstruktion ergeben und so erste erfolgreiche Einsätze in konkreten Industrieprojekten aus den Bereichen der pharmazeutischen Produktion, der Automobilindustrie und der Finanzdienstleistungen ermöglicht hat. Die entsprechend positive Resonanz bestärkt den Autor in der Hoffnung, aus der Sicht der Praktischen Informatik einen sinnvollen Beitrag zu einer effektiveren Softwareentwicklung geleistet zu haben.

[1] Commercial-Off-The-Shelf

A DTDs der XML-Deskriptoren

Die folgenden Document Type Definitions (DTDs) repräsentieren die wesentlichen Aspekte des Typ- bzw. Metainformations-Modells der Komponentensicht der vorliegenden Arbeit. Ein vollständige Darstellung aller beteiligten Deskriptoren und deren Elemente findet sich in [Bernhardt & Djokić, 2001].

A.1 corbacomponent.dtd

```
<!ELEMENT corbacomponent (interface*, componentfeatures+)>
<!ATTLIST corbacomponent
   displayname CDATA #REQUIRED
   componenttypename CDATA #REQUIRED>

<!ELEMENT interface EMPTY>
<!ATTLIST interface
   name CDATA #REQUIRED
   repid ID #REQUIRED>

<!ELEMENT componentfeatures (inheritscomponent?, supportinterface*,
constraint*, ports, extension*)>
<!ATTLIST componentfeatures
   name CDATA #REQUIRED
   repid ID #REQUIRED>

<!ELEMENT inheritscomponent EMPTY>
<!ATTLIST inheritscomponent
   repid IDREF #REQUIRED>

<!ELEMENT supportsinterface EMPTY>
<!ATTLIST supportsinterface
   repid IDREF #REQUIRED>

<!ELEMENT ports (uses | provides | emits | consumes)*>

<!ELEMENT extension (#PCDATA)>
<!ATTLIST extension
   class CDATA #REQUIRED
   origin CDATA #REQUIRED
   id #IMPLIED
   extra CDATA #IMPLIED
   html-form CDATA #IMPLIED>

<!ELEMENT uses EMPTY>
<!ATTLIST uses
   usesname CDATA #REQUIRED
   repid IDREF #REQUIRED>
```

```
<!ELEMENT provides EMPTY>
<!ATTLIST provides
   providesname CDATA #REQUIRED
   repid IDREF #REQUIRED>

<!ELEMENT emits EMPTY>
<!ATTLIST emits
   emitsname CDATA #REQUIRED
   eventtype CDATA #REQUIRED
   instance CDATA #REQUIRED>

<!ELEMENT consumes EMPTY>
<!ATTLIST consumes
   consumesname CDATA #REQUIRED
   eventtype CDATA #REQUIRED
   instance CDATA #REQUIRED>

<!ELEMENT constraint EMPTY>
<!ATTLIST constraint
   constraintname CDATA #REQUIRED>
```

DTD des `CorbaComponentDescriptor`

Die hier angegebene `corbacomponent.dtd` stellt einen Auszug des von der OMG spezifizierten Meta-Modells des CORBA Component Model [OMG, 1999] dar, das dabei um die folgenden XML-Elemente erweitert wurde:

- Angabe des Display Name `displayname` sowie des Komponententypnamens `componenttypename`
- Möglichkeit der Definition von Constraints mittels `constraint`
- Angabe der syntaktischen Realisierung `instance` zu jeder Domänenaktivität (Business Event)

A.2 syntBEType.dtd

```
<!ELEMENT syntBEType (semMessageTypes)>
<!ATTLIST syntBEType
   typename CDATA #REQUIRED>

<!ELEMENT semMessageTypes (semMessageType+)>

<!ELEMENT semMessageType (parameter, returnvalue)>
<!ATTLIST semMessageType
   typename CDATA #REQUIRED
   instance CDATA #REQUIRED>

<!ELEMENT parameter (datatype*)>

<!ELEMENT returnvalue (datatype?)>

<!ELEMENT datatype EMPTY>
<!ATTLIST datatype
   type CDATA #REQUIRED>
```

DTD des `SyntBETypeDescriptor`

A.3 syntMessageType.dtd

```
<!ELEMENT syntMessageType (OperationTypes, CommunicationType,
CommunicationModelType)>
<!ATTLIST syntMessageType
   typename CDATA #REQUIRED>

<!ELEMENT OperationTypes (OperationType+)>

<!ELEMENT OperationType (parameter, returnvalue)>
<!ATTLIST OperationType
   typename CDATA #REQUIRED>

<!ELEMENT parameter (datatype*)>

<!ELEMENT returnvalue (datatype?)>

<!ELEMENT datatype EMPTY>
<!ATTLIST datatype
   type CDATA #REQUIRED>

<!ELEMENT CommunicationType EMPTY>
<!ATTLIST CommunicationType
   value (Synchron | Asynchron | Disconnected) ”Synchron”>

<!ELEMENT CommunicationModelType EMPTY>
<!ATTLIST CommunicationModelType
   value (EventModel | MethodCallModel | MessageQueueModel)
   ”MethodCallModel”>
```

DTD des `SyntMessageTypeDescriptor`

A.4 FeatureStructure.dtd

```
<!ELEMENT FeatureStructure (feature)>
<!ATTLIST FeatureStructure
   typename CDATA #REQUIRED>

<!ELEMENT feature (feature*)>
<!ATTLIST feature
   name CDATA #REQUIRED
   datatype CDATA #REQUIRED>
```

DTD der Merkmalsstrukturbeschreibung

Auf der `FeatureStructure.dtd` basieren der `SemBEType-Descriptor` sowie der `ConstraintDescriptor`.

A.5 NFA.dtd

```
<!ELEMENT NFA (states, alphabet, edges, start, ends)>
<!ATTLIST NFA
```

```
   reference CDATA #REQUIRED
   name CDATA #REQUIRED>

<!ELEMENT states (state+)>

<!ELEMENT state EMPTY>
<!ATTLIST state
   name CDATA #REQUIRED
   id ID #REQUIRED>

<!ELEMENT alphabet (element+)>

<!ELEMENT element EMPTY>
<!ATTLIST element
   name CDATA #REQUIRED
   id ID #REQUIRED>

<!ELEMENT edges (edge+)>

<!ELEMENT edge EMPTY>
<!ATTLIST edge
   prestate IDREF #REQUIRED
   element IDREF #REQUIRED
   poststate IDREF #REQUIRED>

<!ELEMENT start EMPTY>
<!ATTLIST start
   id IDREF #REQUIRED>

<!ELEMENT ends (end+)>

<!ELEMENT end EMPTY>
<!ATTLIST end
   id IDREF #REQUIRED>
```

DTD der Protokollbeschreibung

Die `NFA.dtd` liegt dem `BEProtocolDescriptor`, dem `MessageProtocolDescriptor` sowie dem `OperationProtocolDescriptor` zu Grunde.

B Formale Grundlagen der Protokollkompatibilität

Da der interaktionsorientierte bzw. u.U. adaptive Kopplungsvorgang von zentraler Bedeutung für eine korrekte Applikationsgenerierung im Sinne dieser Arbeit ist, sollte der zu Grunde gelegte Kompatibilitätsbegriff möglichst formal abgesichert sein. Eine entsprechende Betrachtung wird im Folgenden vorgenommen.

Die Darstellung bezieht sich dabei auf die in den Abschnitten 3.1.1 und 3.6 vorgenommenen Betrachtungen zur Komponentenkompatibilität und deren Prüfung. Es werden formale Definitionen der Begriffe *unspezifizierter Empfang*, *Verklemmungsfreiheit*, und *Kompatibilität* präsentiert. Zudem wird die Basis des in Abschnitt 3.6.2 präsentierten Algorithmus hergeleitet und bewiesen. Ein alternativer Algorithmus zur Kompatibilitätsprüfung, der das Auftreten von Schleifen im Kommunikationsablauf (siehe Abschnitt 3.4.7) nachzuweisen in der Lage ist, wird am Ende dieses Anhangs diskutiert.

Die Zustände eines Protokolls `P` werden als `States(P)` und seine Zustandsübergänge als `Transitions(P)` bezeichnet. Für zwei kollaborierende Protokolle $\mathtt{P}_1$ und $\mathtt{P}_2$ wird angenommen, dass die Bezeichner der Nachrichten, die von $\mathtt{P}_1$ zu $\mathtt{P}_2$ gesendet werden, sich von den Bezeichnern der Nachrichten, die von $\mathtt{P}_2$ zu $\mathtt{P}_1$ gesendet werden, unterscheiden. Diese Annahme erlaubt die Schreibweise eines Zustandsüberganges als $\mathtt{s} : \mathtt{m} \mapsto \mathtt{s}'$ – also das Weglassen der »+« bzw. »–« -Zeichen. Die Hilfsfunktion `Polarity(P,m)` wird gleich »+« sein, wenn `m` eine zu empfangende Nachricht im Protokoll `P` ist, und »–« , wenn `m` eine zu sendende Nachricht im Protokoll `P` ist.

Ein *Kollaborationszustand* α für zwei Protokolle $\mathtt{P}_1$ und $\mathtt{P}_2$ ist ein Tupel `(s,t,m)`, wobei $\mathtt{s} \in \mathtt{States}(\mathtt{P}_1)$, $\mathtt{t} \in \mathtt{States}(\mathtt{P}_2)$, und `m` entweder Bezeichner einer Nachricht oder das spezielle Symbol $\perp$ ist. Eine *Kollaborationshistorie* für zwei Protokolle $\mathtt{P}_1$ und $\mathtt{P}_2$ ist eine potentiell unendliche Sequenz von Kollaborationszuständen $\alpha_1, \alpha_2, \ldots$, wobei

- $\alpha_1 = (\mathtt{init}_{P1}, \mathtt{init}_{P2}, m_1)$,

- $\alpha_{i+1} = (s_{i+1}, t_{i+1}, m_{i+1})$, wenn $\alpha_i = (s_i, t_i, m_i)$ und $s_i : m_i \mapsto s_{i+1} \in \mathtt{Transitions}(P_1), t_i : m_i \mapsto t_{i+1} \in \mathtt{Transitions}(P_2)$ und $\mathtt{Polarity}(P_1, m_i) \neq \mathtt{Polarity}(P_2, m_i)$.

- Wenn die Kollaborationshistorie endlich ist, dann besitzt der letzte Kollaborationszustand dieser Kollaborationshistorie die Form $(s_n, t_n, \perp)$.

Des Weiteren wird definiert: $\mathtt{Collabs}(P_1, P_2) = \{\alpha : \alpha$ ist eine Kollaborationshistorie für P_1 und $P_2\}$. Erwähnenswert ist, dass jedes Präfix einer Kollaborationshistorie in eine Kollaborationshistorie umgewandelt werden kann, indem beim letzten Kollaborationszustand des Präfix der Bezeichner der Nachricht durch das Symbol $\perp$ ersetzt wird.

Zwei Protokolle P_1 und P_2 haben keinen *unspezifizierten Empfang*, wenn für alle $\alpha \in \mathtt{Collabs}(P_1, P_2)$, für alle $n, \alpha = \ldots, \alpha_n, \ldots$, wobei $\alpha_n = (s_n, t_n, m_n)$, gilt:

- Wenn $s = s_n$, $s : m \mapsto s' \in \mathtt{Transitions}(P_1)$ und $\mathtt{Polarity}(P_1, m) = -$, dann existiert $t : m \mapsto t' \in \mathtt{Transitions}(P_2)$, $t = t_n$, so dass $\mathtt{Polarity}(P_2, m) = +$, und

- wenn $t = t_n$, $t : m \mapsto t' \in \mathtt{Transitions}(P_2)$ und $\mathtt{Polarity}(P_12, m) = -$, dann existiert $s : m \mapsto s' \in \mathtt{Transitions}(P_1)$, $s = s_n$, so dass $\mathtt{Polarity}(P_1, m) = +$.

Das bedeutet, dass P_1 und P_2 keinen unspezifizierten Empfang haben, wenn gilt: Immer wenn eine Kollaboration den Punkt erreicht, wo $P_1(P_2)$ in einem Zustand ist, in dem eine Nachricht m gesendet werden kann, dann ist der Kollaborationspartner in einem Zustand, in dem er diese Nachricht empfangen kann.

Zwei Protokolle P_1 und P_2 sind *verklemmungsfrei*, wenn für alle endlichen Sequenzen $\alpha \in \mathtt{Collabs}(P_1, P_2)$, $\alpha = \alpha_1, \ldots, \alpha_{n-1}, \alpha_n$, wobei $\alpha_n = (s_n, t_n, \perp)$ gilt: Entweder

- s_n und t_n sind Endzustände von P_1 und P_2, oder

- es existiert eine Nachricht m, so dass (i) $s : m \mapsto s' \in \mathtt{Transitions}(P_1)$, wobei $s = s_n$, (ii) $t : m \mapsto t' \in \mathtt{Transitions}(P_2)$, wobei $t = t_n$, und (iii) $\mathtt{Polarity}(P_1, m) \neq \mathtt{Polarity}(P_2, m)$.

Das bedeutet, dass $\mathtt{P}_1$ und $\mathtt{P}_2$ verklemmungsfrei sind, wenn für jede Kollaboration gilt, dass die Kollaboration entweder in Endzuständen endet oder fortgesetzt werden kann.

Zwei Protokolle $\mathtt{P}_1$ und $\mathtt{P}_2$ sind *kompatibel*, wenn $\mathtt{Collabs}(\mathtt{P}_1, \mathtt{P}_2) \neq \emptyset$, sie verklemmungsfrei sind und kein unspezifizierter Empfang auftritt.

Eine Überprüfung dieses Kriteriums der Protokollkompatibilität erfordert eigentlich die Untersuchung aller (potenziell unendlicher) Kollaborationshistorien. Zur Lösung dieser Problematik wird im Folgenden eine Umformulierung vorgenommen, die einen Algorithmus zur Kompatibilitätsprüfung ermöglicht, der nicht auf der Untersuchung potenziell unendlicher Sequenzen basiert.

Es sei $\mathtt{s} \in \mathtt{States}(\mathtt{P}_1)$ und $\mathtt{t} \in \mathtt{States}(\mathtt{P}_2)$. Per Definition gilt $\mathtt{s} \sim \mathtt{t}$, wenn eine Kollaborationshistorie $\alpha = \ldots \alpha_\mathtt{i} \ldots \in \mathtt{Collabs}(\mathtt{P}_1, \mathtt{P}_2)$ mit $\alpha_\mathtt{i} = (\mathtt{s}, \mathtt{t}, \mathtt{m})$ für ein $\mathtt{m}$ existiert.

Es sei $\mathtt{EQUIV}(\mathtt{P}_1, \mathtt{P}_2) \subseteq \mathtt{States}(\mathtt{P}_1) \times \mathtt{States}(\mathtt{P}_2)$ die kleinste Menge, für die gilt:

- $(\mathtt{init}_{\mathtt{P1}}, \mathtt{init}_{\mathtt{P2}}) \in \mathtt{EQUIV}(\mathtt{P}_1, \mathtt{P}_2)$, und
- wenn $(\mathtt{s}, \mathtt{t}) \in \mathtt{EQUIV}(\mathtt{P}_1, \mathtt{P}_2)$, $\mathtt{s} : \mathtt{m} \mapsto \mathtt{s}' \in \mathtt{Transitions}(\mathtt{P}_1)$, $\mathtt{t} : \mathtt{m} \mapsto \mathtt{t}' \in \mathtt{Transitions}(\mathtt{P}_2)$ und $\mathtt{Polarity}(\mathtt{P}_1, \mathtt{m}) \neq \mathtt{Polarity}(\mathtt{P}_2, \mathtt{m})$, dann $(\mathtt{s}', \mathtt{t}') \in \mathtt{EQUIV}(\mathtt{P}_1, \mathtt{P}_2)$.

Die Menge $\mathtt{EQUIV}(\mathtt{P}_1, \mathtt{P}_2)$ ist offensichtlich endlich und kann durch einen Algorithmus erzeugt werden.

Folgende Annahme ermöglicht nun eine Verwendung dieses Algorithmus zur Prüfung der Kompatibilität zweier Protokolle:

- $(\mathtt{s}, \mathtt{t}) \in \mathtt{EQUIV}(\mathtt{P}_1, \mathtt{P}_2)$, wenn $\mathtt{s} \sim \mathtt{t}$.

Diese Annahme läßt sich mit folgendem Beweis rechtfertigen: Für alle $\mathtt{s}$ und $\mathtt{t}$ mit $\mathtt{s} \sim \mathtt{t}$ sei $\mathtt{len}(\mathtt{s} \sim \mathtt{t})$ wenn (I) ein $\alpha = \alpha_1, \ldots, \alpha_\mathtt{i}, \ldots \in \mathtt{Collabs}(\mathtt{P}_1, \mathtt{P}_2)$ mit $\alpha_\mathtt{i} = (\mathtt{s}, \mathtt{t}, \mathtt{m}_\mathtt{i})$ existiert und (II) falls ein $\beta = \beta_1, \ldots, \beta_\mathtt{i}, \ldots \in \mathtt{Collabs}(\mathtt{P}_1, \mathtt{P}_2)$ mit $\beta_\mathtt{j} = (\mathtt{s}, \mathtt{t}, \mathtt{m}_\mathtt{j})$ existiert, dann ist $\mathtt{j} \geq \mathtt{i}$. Es läßt sich nun durch einen Induktionsbeweis auf $\mathtt{len}(\mathtt{s} \sim \mathtt{t})$ leicht zeigen, dass $(\mathtt{s}, \mathtt{t}) \in \mathtt{EQUIV}(\mathtt{P}_1, \mathtt{P}_2)$, wenn $\mathtt{s} \sim \mathtt{t}$.

- Induktionsanfang: Für $\mathtt{len}(\mathtt{s} \sim \mathtt{t}) = 1$ besteht die Kollaborationshistorie lediglich aus $\alpha_1 = (\mathtt{init}_{\mathtt{P1}}, \mathtt{init}_{\mathtt{P2}}, \mathtt{m}_1)$. Es gilt also $\mathtt{s} = \mathtt{init}_{\mathtt{P1}}$ und $\mathtt{t} = \mathtt{init}_{\mathtt{P2}}$. Per Definition ist auch $(\mathtt{init}_{\mathtt{P1}}, \mathtt{init}_{\mathtt{P2}}) \in \mathtt{EQUIV}(\mathtt{P}_1, \mathtt{P}_2)$.

- Induktionsschritt: Aus $\mathtt{s} \sim \mathtt{t}$ mit $\mathtt{len}(\mathtt{s} \sim \mathtt{t}) = \mathtt{i} - 1$ läßt sich genau dann ein $\mathtt{s}' \sim \mathtt{t}'$ mit $\mathtt{len}(\mathtt{s}' \sim \mathtt{t}') = \mathtt{i}$ herleiten, wenn eine Nachricht $\mathtt{m_i}$ existiert, sodass ein $\alpha_\mathtt{i} = (\mathtt{s}', \mathtt{t}', \mathtt{m_i})$ aus einem $\alpha_{\mathtt{i}-1} = (\mathtt{s}, \mathtt{t}, \mathtt{m_{i-1}})$ unter den Bedingungen $\mathtt{s} : \mathtt{m_i} \mapsto \mathtt{s}' \in \mathtt{Transitions}(\mathtt{P_1}), \mathtt{t} : \mathtt{m_i} \mapsto \mathtt{t}' \in \mathtt{Transitions}(\mathtt{P_2})$ und $\mathtt{Polarity}(\mathtt{P_1}, \mathtt{m_i}) \neq \mathtt{Polarity}(\mathtt{P_2}, \mathtt{m_i})$ erreicht werden kann (und $(\mathtt{s}', \mathtt{t}', \mathtt{m_i})$ nicht Teil der Kollaborationshistorie $\alpha_{\mathtt{i}-1}$ ist). Wenn eine derartige Nachricht $\mathtt{m_i}$ existiert, so ist per Definition jedoch auch $(\mathtt{s}', \mathtt{t}') \in \mathtt{EQUIV}(\mathtt{P_1}, \mathtt{P_2})$ (siehe Definition der Menge EQUIV). □

Die Relation $\sim$ erfasst genau die Zustände $\mathtt{s}, \mathtt{t} \in \mathtt{States}(\mathtt{P_1}) \times \mathtt{States}(\mathtt{P_2})$, die gemeinsam in einem Kollaborationszustand aus $\mathtt{Collabs}(\mathtt{P_1}, \mathtt{P_2})$ auftreten können. Somit ist ein Algorithmus, der genau diese Zustände sammelt und eine Untersuchung ermöglicht, ohne die Untersuchung potenziell unendlicher Sequenzen zu benötigen, implementierbar. Ein derartiger Algorithmus findet sich in Abschnitt 3.6.2. Basis dieses Algorithmus sind Elemente `(s,t)` der Menge $\mathtt{EQUIV}(\mathtt{P_1}, \mathtt{P_2})$.

Eine alternative Basis für einen Algorithmus – ebenfalls unter Vermeidung der Untersuchung potenziell unendlicher Sequenzen – wäre eine Menge $\mathtt{EQUIVCOL}(\mathtt{P_1}, \mathtt{P_2})$, die aus allen Kollaborationshistorien besteht, in denen zwei Zustände $\mathtt{s}, \mathtt{t} \in \mathtt{States}(\mathtt{P_1}) \times \mathtt{States}(\mathtt{P_2})$ maximal einmal gemeinsam in einem Kollaborationszustand vorkommen dürfen.

Die Menge $\mathtt{EQUIVCOL}(\mathtt{P_1}, \mathtt{P_2})$ ist ebenfalls endlich und läßt sich unter leichter Veränderung des Algorithmus aus Abschnitt 3.6.2 erzeugen. Ein derartiger Algorithmus läßt sich ebenfalls zur Kompatibilitätsprüfung verwenden und darüber hinaus läßt sich die Existenz von Schleifen (siehe Abschnitt 3.4.7) nachweisen: Wenn bei der Suche nach Nachfolgezuständen einer Kollaborationshistorie ein Kollaborationszustand erreicht wird, der in der Kollaborationshistorie bereits vorkommt, wird eine Schleife vermerkt. Ein derartiger Algorithmus wurde im Rahmen der vorligenden Arbeit ebenfalls prototypisch umgesetzt.

C Beispiel-Protokolle des Bankszenarios

C.1 Kundenprotokoll

***Listing C.1**: Bankprotokoll*

```
1: -start              -> 2;   ()
2: +authenticate       -> 3;   ()
2: +abort              -> 15;  (string reason)
3: -id                 -> 4;   (int account, int pin)
3: +abort              -> 15;  (string reason)
4: +accessGranted      -> 5;   (int sessionId)
4: +abort              -> 15;  (string reason)
5: -showAccount        -> 6;   (int sessionId)
5: -withdraw           -> 7;   (int sessionId, real value)
5: -deposit            -> 8;   (int sessionId, real value)
5: -remittanceStart    -> 9;   (int sessionId, real value, int blz,
                                int targetAccount)
5: -comission          -> 12;  (int sessionId, real value, int blz,
                                int targetAccount, int date_of_month)
5: -stop               -> 1;   (int sessionId)
5: +abort              -> 15;  (string reason)
6: +account            -> 5;   (real value)
7: +withdrawDone       -> 5;   (real newValue)
7: +withdrawReject     -> 5;   (string reason, real oldValue)
8: +depositDone        -> 5;   (real newValue)
8: +depositReject      -> 5;   (string reason, real oldValue)
9: +remittanceOk       -> 10;  (int remittanceId)
9: +remittanceNotOk    -> 5;   (string reason)
10: -doRemit           -> 11;  (int remittanceId)
10: -remittanceAbort   -> 5;   (int remittanceId)
11: +remittanceDone    -> 5;   (int remittanceId, real newValue)
11: +remittanceReject  -> 5;   (int remittanceId, real oldValue)
12: +comissionDone     -> 5;   ()
12: +comissionReject   -> 5;   (string reason)
15: -abortConfirm      -> 1;   ()
```

C.2 Bankprotokoll

***Listing C.2**: Bankprotokoll*

```
1: +start              -> 2;    ()
2: -authenticate       -> 3;    ()
2: -abort              -> 15;   (string reason)
3: +id                 -> 4;    (int account, int pin)
3: -abort              -> 15;   (string reason)
4: -accessGranted      -> 5;    (int sessionId)
4: -abort              -> 15;   (string reason)
5: +showAccount        -> 6;    (int sessionId)
5: +withdraw           -> 7;    (int sessionId, real value)
5: +deposit            -> 8;    (int sessionId, real value)
5: +remittanceStart    -> 9;    (int sessionId, real value, int blz,
                                 int targetAccount)
5: +comission          -> 12;   (int sessionId, real value, int blz,
                                 int targetAccount, int date_of_month)
5: +stop               -> 1;    (int sessionId)
5: -abort              -> 15;   (string reason)
6: -account            -> 5;    (real value)
7: -withdrawDone       -> 5;    (real newValue)
7: -withdrawReject     -> 5;    (string reason, real oldValue)
8: -depositDone        -> 5;    (real newValue)
8: -depositReject      -> 5;    (string reason, real oldValue)
9: -remittanceOk       -> 10;   (int remittanceId)
9: -remittanceNotOk    -> 5;    (string reason)
10: +doRemit           -> 11;   (int remittanceId)
10: +remittanceAbort   -> 5;    (int remittanceId)
11: -remittanceDone    -> 5;    (int remittanceId, real newValue)
11: -remittanceReject -> 5;     (int remittanceId, real oldValue)
12: -comissionDone     -> 5;    ()
12: -comissionReject   -> 5;    (string reason)
15: +abortConfirm      -> 1;    ()
```

C.3 Sparkassenprotokoll

***Listing C.3**: Sparkassenprotokoll*

```
1: +begin          -> 2;  (int acc, int pin)
2: -access         -> 3;  ()
2: -accessDenied   -> 1;  (string reason)
3: +showFile       -> 4;  (int acc)
3: +withdraw       -> 5;  (int acc, real money)
3: +deposit        -> 6;  (int acc, real money)
3: +remittance     -> 7;  (int acc, real money, int blz, int target)
3: +end            -> 1;  ()
4: -file           -> 3;  (real money)
5: -withdrawDone   -> 3;  (real new)
5: -noAction       -> 3;  (string reason, real money)
6: -depositDone    -> 3;  (real new)
6: -noAction       -> 3;  (string reason, real money)
7: -remittanceDone -> 3;  (real new)
7: -noAction       -> 3;  (string reason, real money)
```

C.4 Metaadapterprotokoll zur Zusammenarbeit Kunde / Sparkasse

***Listing C.4**: Protokoll des Metaadapters*

```
//Parameter mapping rules
Kunde::id.account                     -> Sparkasse::begin.acc;
Kunde::id.pin                         -> Sparkasse::begin.pin;
Kunde::id.account                     -> Kunde::accessGranted.sessionId;
Kunde::id.account                     -> Sparkasse::showFile.acc;
Kunde::id.account                     -> Sparkasse::withdraw.acc;
Kunde::withdraw.value                 -> Sparkasse::withdraw.money;
Kunde::id.account                     -> Sparkasse::deposit.acc;
Kunde::deposit.value                  -> Sparkasse::deposit.money;
Kunde::id.account                     -> Sparkasse::remittance.acc;
Kunde::remittanceStart.value          -> Sparkasse::remittance.money;
Kunde::remittanceStart.blz            -> Sparkasse::remittance.blz;
Kunde::remittanceStart.targetAccount -> Sparkasse::remittance.target;

Sparkasse::accesDenied.reason -> Kunde::abort.reason;
Sparkasse::file.money         -> Kunde::account.value;
Sparkasse::withdrawDone.new   -> Kunde::withdrawDone.newValue;
Sparkasse::noAction.reason    -> Kunde::withdrawReject.reason;
Sparkasse::noAction.money     -> Kunde::withdrawReject.oldValue;
Sparkasse::depositDone.new    -> Kunde::depositDone.newValue;
Sparkasse::noAction.reason    -> Kunde::depositReject.reason;
Sparkasse::noAction.money     -> Kunde::depositReject.oldValue;
Sparkasse::remittanceDone.new -> Kunde::remittanceDone.newValue;
Sparkasse::noAction.money     -> Kunde::remittanceReject.oldValue;
Kunde::remittanceStart.blz    -> Kunde::remittanceOk.remittanceId;
"not available"               -> Kunde::comissionReject.reason;

//oneshot-rules
one-shot(Kunde::id.pin);
one-shot(Kunde::withdraw.value);
one-shot(Kunde::deposit.value);
one-shot(Kunde::remittanceStart.value);
one-shot(Kunde::remittanceStart.targetAccount);
one-shot(Sparkasse::accessDenied.reason);
one-shot(Sparkasse::file.money);
one-shot(Sparkasse::withdrawDone.new);
one-shot(Sparkasse::depositDone.new);
one-shot(Sparkasse::remittanceDone.new);

//causality rules
A: +start::kunde               -> B,
B: -authenticate::kunde        -> C;
C: +id::kunde                  -> D;
D: -begin::sparkasse           -> E;
E: +abortConfirm::kunde        -> A;
E: +stop::kunde                -> A;

A: +access::sparkasse          -> B;
B: -accessGranted::kunde       -> A;

A: +accessDenied::sparkasse    -> B;
```

```
B: -abort::kunde                -> A;

A: +showAccount::kunde          -> B;
B: -showFile::sparkasse         -> A;

A: +withdraw::kunde             -> B;
B: -withdraw::sparkasse         -> A;

A: +deposit::kunde              -> B;
B: -deposit::sparkasse          -> A;

A: +stop::kunde                 -> B;
B: -end::sparkasse              -> A;

A: +file::sparkasse             -> B;
B: -account::kunde              -> A;

A: +withdraw::kunde             -> B;
B: +withdrawDone::sparkasse     -> C;
B: +noAction::sparkasse         -> D;
C: -withdrawDone::kunde         -> A;
D: -withdrawReject::kunde       -> A;

A: +deposit::kunde              -> B;
B: +depositDone::sparkasse      -> C;
B: +noAction::sparkasse         -> D;
C: -depositDone::kunde          -> A;
D: -depositReject::kunde        -> A;

A: +remittanceStart::kunde      -> B;
B: -remittanceOk::kunde         -> C;
C: +doRemit::kunde              -> D;
D: -remittance::sparkasse       -> E;
E: +remittanceDone::sparkasse   -> F;
E: +noAction::sparkasse         -> G;
F: -remittanceDone::kunde       -> A;
G: -remittanceReject::kunde     -> A;

A: +comission::kunde            -> B;
B: -comissionReject::kunde      -> A;
```

D Definition exemplarischer Komponententypen

Dieser Anhang diskutiert zur Illustration des dieser Arbeit zu Grunde liegenden Typmodells zwei im Rahmen von [Häming, 2000] entworfene Beispielszenarien aus den Domänen *Bankwesen* und *Druckdienste*, an denen die zu Grunde liegenden Konzepte und Mechanismen des Typmanagements konkret demonstriert werden sollen. Für jede Typdomäne werden dabei im ersten Schritt exemplarische Komponententypen definiert, indem entsprechende Typbeschreibungen festgelegt werden, die die Grundlage für die nachfolgende Betrachtung darstellen. Diese legt den Schwerpunkt auf die evaluierten Beziehungen zwischen den Komponententypen im Rahmen von Typvergleichen und Kompatibilitätsprüfungen, wobei zudem auf Adaptionsmechanismen sowie die Realisierung von Rollen eingegangen wird.

D.1 Kunde-Bank-Beispiel

Das folgende Beispiel illustriert ein Szenario dreier Komponententypen *Bank*, *Savings Bank* (Sparkasse) und *Customer* (Kunde), deren Instanzen als Komponenten zur Konstruktion von Anwendungen aus der Domäne der Finanzdienstleistungen aufzufassen sind. Die *Customer*-Komponente nimmt in diesem Beispiel die Funktionalität eines Komponententyps *Bank* in Anspruch, die – um das Beispiel einfach zu halten – lediglich darin besteht, als Kundenzugang beziehungsweise -abgang bei der Bank verzeichnet zu werden, um Überweisungsaufträge vorzunehmen. Der Komponententyp *Savings Bank* ist dabei als eine Typart der *Bank*-Komponente konzipiert, sodass die *Savings Bank* einen potentiellen Substitutionskandidaten für diese darstellt. Im Gegensatz dazu fungiert der Komponententyp *Customer* als möglicher Interaktionspartner.

Der erste Schritt besteht an dieser Stelle in der Festlegung der Typbeschreibungen der beteiligten Komponenten. Dabei wird

auf Grund der nach unten zunehmenden Breite des Typbaums lediglich ein relevanter Querschnitt durch diesen vorgestellt. Auf der zweiten sowie dritten Ebene des Typbaums hat dies die Darstellung lediglich *einer* ausgewählten syntaktischen Realisierung zur Folge.

D.1.1 Typbeschreibungen der Komponenten

In Abbildung D.1 ist zunächst ein Ausschnitt des Typbaums des Komponententyps *Bank* dargestellt, zu dem anschließend potenzielle Substitutionskandidaten sowie Interaktionspartner definiert werden sollen.

Semantisch wird der Komponententyp *Bank* durch das Konsumieren der Business Events *Register* (Kundenzugang), *Unregister* (Kundenabgang) sowie *TransferRequest* (Überweisungsauftrag) charakterisiert, wobei die *Bank* selbst keine Business Events emittiert. Die Semantik der Business Events wird durch Feature Structures repräsentiert. Alle möglichen Sequenzen von Business Events werden durch deterministische endliche Automaten (DFAs) definiert, die die abstrakten Schnittstellen der *Bank* in Form von Protokollen auf der semantischen Ebene darstellen. In diesem Fall wird lediglich ein Protokoll spezifiziert, wobei keine Constraints definiert werden, ausgedrückt durch die leere Constraint-Menge.

Auf der Nachrichtenebene ist exemplarisch die syntaktische Realisierung des Business Events *TransferRequest* dargestellt, die sich aus zwei semantischen Nachrichtentypen *prüfeKunde* und *überweise* zusammensetzt. Die semantischen Nachrichtentypen approximieren dabei den syntaktischen Typ des Business Events durch jeweils eine Menge von Parametertypen mit einem Rückgabetyp, wobei der Parametertyp `CommunicatorInterface` als eine allgemeine Komponentenschnittstelle des Interaktionspartners aufzufassen ist. Analog zur semantischen Ebene werden alle gültigen Nachrichtensequenzen über deterministische endliche Automaten festgelegt.

Eine konkrete syntaktische Realisierung – hier exemplarisch die des Nachrichtentyps *überweise* – wird auf der syntaktischen Ebene angegeben. Diese beinhaltet zwei Operationstypen *debit* (abbuchen) und *deposit* (einzahlen), die Signaturen – von in diesem Fall entsprechenden Methoden – beschreiben. Deterministische endliche Automaten spezifizieren wiederum die möglichen Sequenzen von Operationsaufrufen. Weitere syntaktische Metainformationen auf dieser Ebene stellen der Kommunikationstyp – hier *synchron* – sowie der Kommunika-

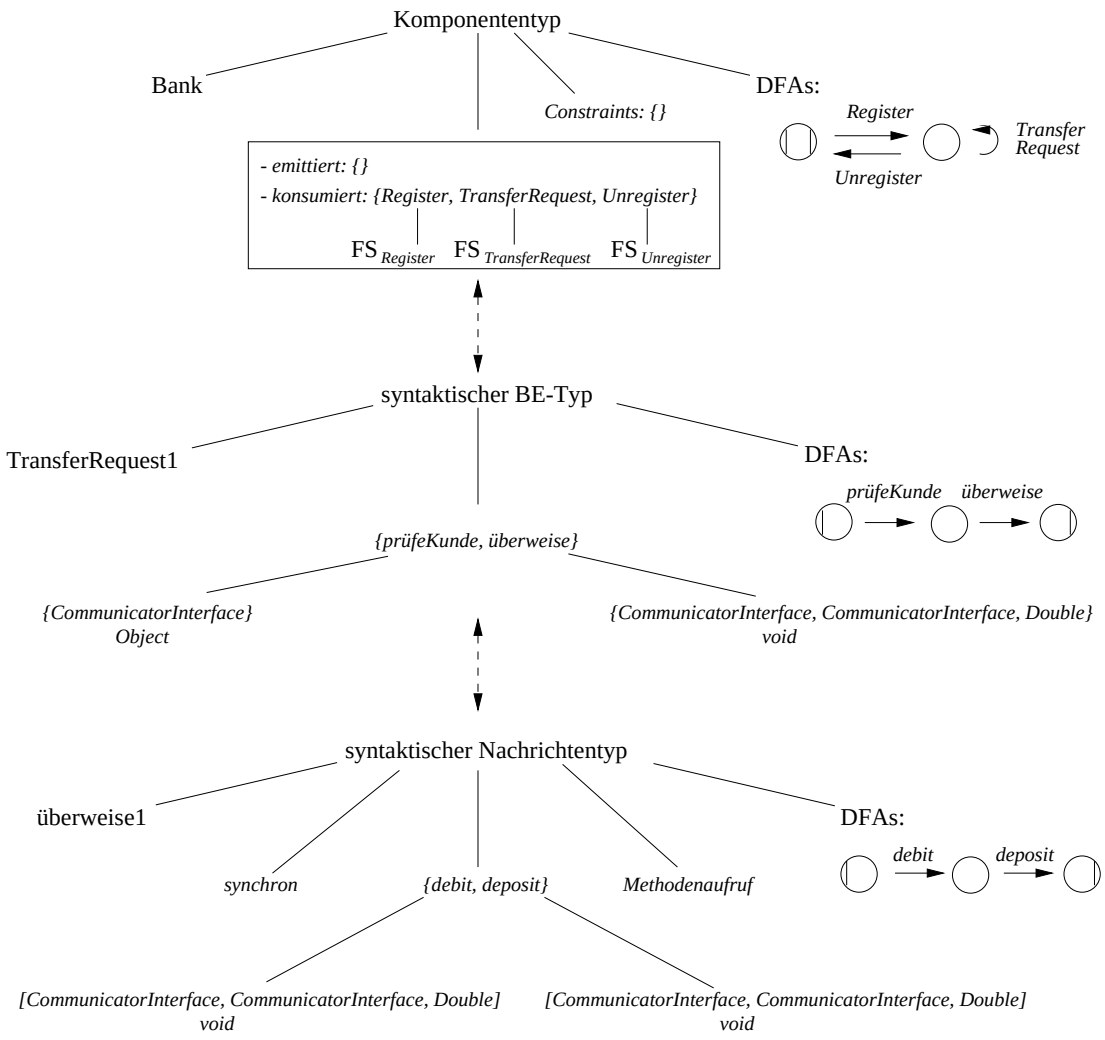


Abbildung D.1
Komponententyp Bank

tionsmodelltyp – in diesem Fall *Methodenaufruf* – dar.

Im Folgenden wird nun die Typart *Savings Bank* definiert, die im Hinblick auf ihre Substitutionseigenschaft zur Ersetzung der *Bank*-Komponente im Rahmen eines Typvergleichs analysiert werden kann. In diesem Zusammenhang kann eine Aussage über die Konformität der *Savings Bank* zu ihrem Basistyp auf den drei Ebenen des Typbaums abgeleitet werden.

D.1.1.1 Substitutionskandidaten

Zur Durchführung eines exemplarischen Typvergleichs wird an dieser Stelle ein Komponententyp *Savings Bank* definiert, der als Typart des Basistyps *Bank* aufzufassen ist. Einen Auszug des Typbaums der *Savings Bank* zeigt Abbildung D.2.

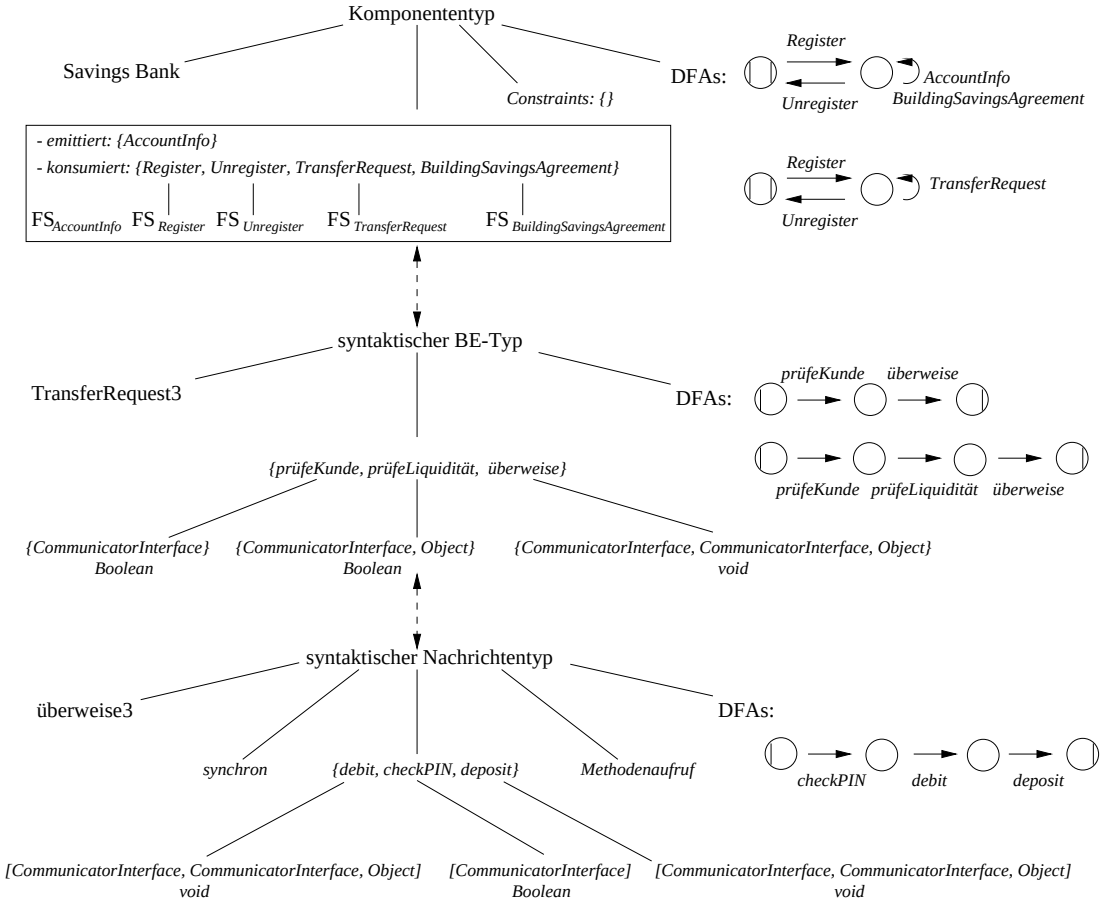


Abbildung D.2
Komponententyp
Savings Bank

Auf der semantischen Ebene konsumiert der Komponententyp *Savings Bank* zunächst alle auch von der *Bank* konsumierten Business Events. Für diese Teilmenge der Business Events besitzt die *Savings Bank* ein zur *Bank*-Komponente identisches Protokoll. Somit trägt die *Savings Bank* dem Interaktionsmuster ihres Basistyps Rechnung, indem sich diese mindestens auf das (eingehende sowie das ausgehende) Interaktionspotenzial der *Bank*-Komponente stützt. Ferner definiert die *Savings Bank* ein zusätzliches Protokoll, das zwei weitere Business Events *Ac-*

countInfo (Kontoauszug) und *BuildingSavingsAgreement* (Bausparvertrag) einführt. Das Business Event *AccountInfo* wird dabei von dem Komponententyp emittiert, während das Business Event *BuildingSavingsAgreement* konsumiert werden kann. Die Business-Event-Semantik wird abermals durch Feature Structures repräsentiert. Letztendlich werden für die *Savings Bank* keine typrelevanten Regeln festgelegt, ausgedrückt durch die leere Constraint-Menge.

Die syntaktische Realisierung des Business Events *TransferRequest* definiert auf der Nachrichtenebene – im Vergleich zu ihrer Entsprechung auf Seiten der *Bank* – einen zusätzlichen semantischen Nachrichtentyp *prüfeLiquidität* sowie subtypisiert die der *Bank* zu Grunde liegenden semantischen Nachrichtentypen *prüfeKunde* und *überweise*, wobei die Subtypisierung durch die Definition kontravarianter Parametertypen mit kovariantem Rückgabetyp realisiert wird. In diesem Fall fordert die *Savings Bank* – ebenso wie die *Bank*-Komponente – als Parametertyp die allgemeine Komponentenschnittstelle `CommunicatorInterface`. Ferner liegen auf Seiten der *Savings Bank* ein zur *Bank* identisches sowie ein weiteres Protokoll über die entsprechenden Nachrichten zu Grunde.

Die angegebene syntaktische Realisierung des Nachrichtentyps *überweise* spezifiziert auf der syntaktischen Ebene ein von der *Bank*-Komponente abweichendes Protokoll (Operationstyp *checkPIN*). Des Weiteren ist der syntaktische Nachrichtentyp – im Vergleich zur korrespondierenden Realisierung auf *Bank*-Seite – bezüglich eines Operationstyps *checkPIN* erweitert wie auch hinsichtlich der Operationstypen *debit* sowie *deposit* subtypisiert, da abermals kontravariante Parametertypen sowie kovariante Rückgabetypen vorliegen. Letztendlich werden der Kommunikationstyp auf *synchron* sowie der Kommunikationsmodelltyp auf *Methodenaufruf* festgelegt, sodass sich in diesem Punkt eine Übereinstimmung mit der *Bank* feststellen lässt.

Der hier angegebene Querschnitt des Typbaums korrespondiert dabei zu dem bereits vorgestellten Auszug der *Bank*. Auf Basis dieser Darstellung kann nun ein Typvergleich von *Bank* und *Savings Bank* erfolgen, wobei ferner Aussagen über die Konformität der *SavingsBank* zu ihrem Basistyp *Bank* auf den einzelnen Ebenen des Typbaums abgeleitet werden können. Zur Vervollständigung eines praxisnahen Szenarios wird schließlich noch ein Komponententyp benötigt, der als potenzieller Interaktionspartner fungiert. Auf diesen wird im Folgenden eingegangen.

D.1.1.2 Interaktionspartner

Zur Durchführung einer exemplarischen Kompatibilitätsprüfung soll nachfolgend der Komponententyp *Customer* betrachtet werden. Die *Customer*-Komponente fungiert in diesem Zusammenhang vorerst als Interaktionspartner der *Bank*. Zudem kann aber ebenso die Kompatibilität zur *Savings Bank* analysiert werden, da diese als potenzieller Substitutionskandidat für die *Bank*-Komponente konzipiert wurde. Der Komponententyp *Customer* basiert schließlich auf dem in Abbildung D.3 dargestellten Typbaum-Ausschnitt, der auf der zweiten sowie dritten Ebene zu den bereits vorgestellten Auszügen der *Bank* wie auch der *Savings Bank* korrespondiert.

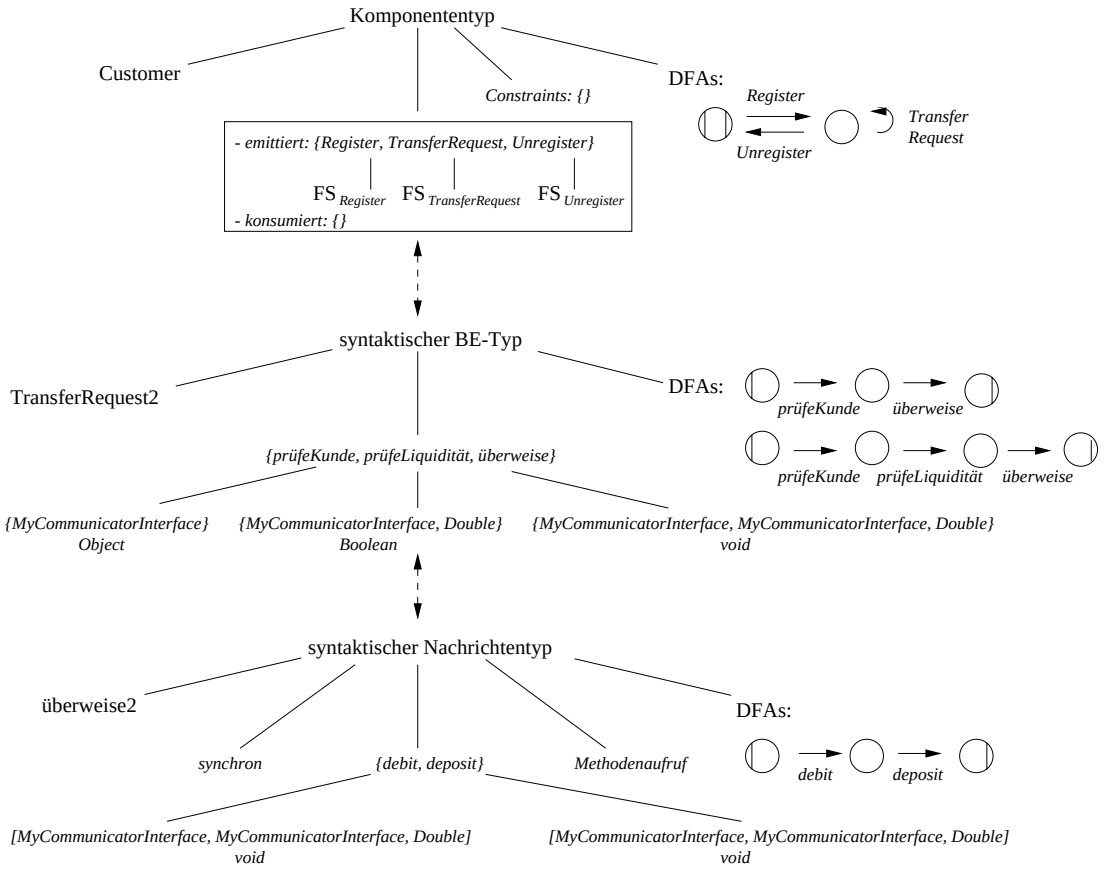


Abbildung D.3
Komponententyp
Customer

Als potenzieller Interaktionspartner der *Bank* zeichnet sich der Komponententyp *Customer* auf semantischer Ebene durch das Emittieren der Business Events *Register*, *Unregister* sowie

TransferRequest aus, wobei ein zur *Bank* identisches Protokoll auf *Customer*-Seite vorhanden ist. Somit basiert der Komponententyp *Customer* hinsichtlich dieses Protokolls auf einem inversen Interaktionsmuster im Vergleich zur *Bank*. Die definierten Business Events werden wiederum semantisch durch Merkmalsstrukturen repräsentiert. Business Events, die konsumiert werden können, sowie Constraints werden letztendlich nicht spezifiziert, ausgedrückt durch die jeweils leere Menge.

Auf der Nachrichtenebene liegt der *Customer*-Komponente eine von der *Bank* verschiedene syntaktische Realisierung des Business Events *TransferRequest* zugrunde. Der syntaktische Business-Event-Typ auf Seiten des *Customer* materialisiert sich dabei in einem zusätzlichen semantischen Nachrichtentyp *prüfeLiquidität* sowie in den der *Bank* zu Grunde liegenden semantischen Nachrichtentypen *prüfeKunde* und *überweise*, deren strukturelle Beschreibungen jeweils in einer Supertypbeziehung zu ihren Entsprechungen auf *Bank*-Seite stehen. Der Parametertyp `MyCommunicatorInterface` stellt dabei einen Subtyp des `CommunicatorInterface` der *Bank* dar. Im Hinblick auf das Komponentenverhalten basiert der *Customer* auf einem zur *Bank* identischen sowie einem weiteren Protokoll über die spezifizierten Nachrichten.

Syntaktisch spiegelt sich dies exemplarisch in der Realisierung des Nachrichtentyps *überweise* wider, die sich nur im Hinblick auf die Subtypisierung von Aufrufparametern der Operationstypen *debit* und *deposit* von der *Bank* unterscheidet. Letztendlich sind mit dem Kommunikationstyp – hier *synchron* – sowie dem Kommunikationsmodelltyp – in diesem Fall *Methodenaufruf* – die Attributwerte gegeben, auf denen bereits die *Bank*-Komponente basiert. Zudem liegt auf der Seite des *Customer* ein zur *Bank* identisches Protokoll über Operationsaufrufe zu Grunde.

Vor der Betrachtung der resultierenden Beziehungen zwischen den exemplarisch definierten Komponententypen soll nachfolgend insbesondere noch auf die Semantikrepräsentation der Business Events eingegangen werden.

D.1.2 Repräsentation der Semantik der Business Events

Für die relevanten Business Events der Komponententypen des Beispielszenarios sollen an dieser Stelle die jeweiligen Semantikrepräsentationen vorgestellt werden. Die Semantik der Busi-

ness Events wird dabei über die Struktur von Merkmalsdiagrammen definiert, die zur Repräsentation auf *Feature Structures* [Carpenter, 1992] abgebildet werden (vgl. a. Anhang I.4 und Abschnitt 6.4.3). Die Merkmalsdiagramme stützen sich auf die aus der Domänenanalyse hervorgegangenen standardisierten Merkmale der modellierten Typdomäne, in diesem Fall der des *Bankwesens*.

D.1.2.1 TransferRequest

In Abbildung D.4 ist das Merkmalsdiagramm des Business Events *TransferRequest* dargestellt. Es beschreibt mit seiner Struktur die Semantik eines *Überweisungsauftrags*.

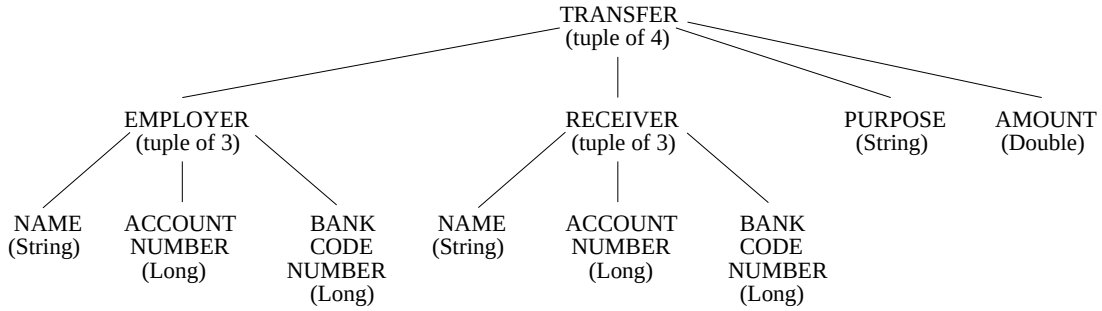


Abbildung D.4 *Merkmalsdiagramm eines* Überweisungsauftrags

In enger Analogie zum Formular eines Überweisungsauftrags der realen Bankwelt zeichnet sich das Merkmalsdiagramm durch ein zusammengesetztes (komplexes) Merkmal (Fachkonzept) *TRANSFER* (Überweisung) aus, das sich aus vier Submerkmalen zusammensetzt, dem *EMPLOYER* (Auftraggeber) und dem *RECEIVER* (Empfänger) – ebenfalls zusammengesetzte Merkmale – sowie den elementaren Merkmalen *PURPOSE* (Verwendungszweck) des Typs `String` und *AMOUNT* (Betrag) des Typs `Double`. Des Weiteren zerlegen sich die zusammengesetzten Merkmale *EMPLOYER* und *RECEIVER* jeweils zu drei elementaren Submerkmale *NAME* (Name) des Typs `String`, *ACCOUNT NUMBER* (Kontonummer) sowie *BANK CODE NUMBER* (Bankleitzahl) des Typs `Long`.

D.1.2.2 Register

Das Business Event *Register* stützt sich auf das in Abbildung D.5 gezeigte Merkmalsdiagramm, das die Semantik eines *Kundenzugangs* repräsentiert.

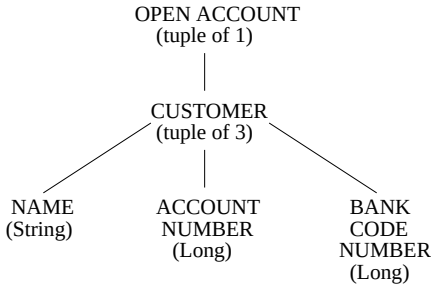


Abbildung D.5
Merkmalsdiagramm eines Kundenzugangs

Das Merkmalsdiagramm des *Kundenzugangs* wird durch ein Merkmal *OPEN ACCOUNT* (Kontoeröffnung) charakterisiert, das ein Submerkmal *CUSTOMER* (Kunde) besitzt, das die drei elementaren Merkmale *NAME* (Name) des Typs `String` sowie *ACCOUNT NUMBER* (Kontonummer) und *BANK CODE NUMBER* (Bankleitzahl) des Typs `Long` subsummiert.

D.1.2.3 Unregister

Für das Business Event *Unregister* wurde das in Abbildung D.6 wiedergegebene Merkmalsdiagramm zu Grunde gelegt. Es stellt die Semantikrepräsentation eines *Kundenabgangs* dar.

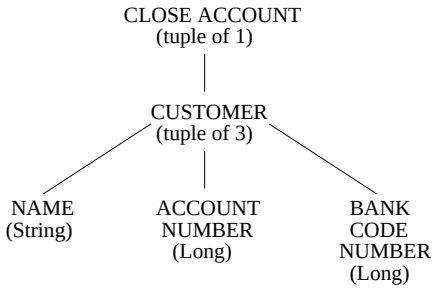


Abbildung D.6
Merkmalsdiagramm eines Kundenabgangs

Das Merkmalsdiagramm des *Kundenabgangs* zeichnet sich durch das Merkmal *CLOSE ACCOUNT* (Kontoschließung) aus, das – analog zu *OPEN ACCOUNT* (Kontoeröffnung) – ein zusammengesetztes Submerkmal *CUSTOMER* spezifiziert, das auf drei elementaren Submerkmalen *NAME* (Name) des Typs `String` wie auch *ACCOUNT NUMBER* (Kontonummer) sowie *BANK CODE NUMBER* (Bankleitzahl) des Typs `Long` basiert.

Im Folgenden können nun die resultierenden Beziehungen

zwischen den Komponententypen *Bank*, *Savings Bank* und *Customer* analysiert werden, wobei die automatische Evaluierung dieser Relationen von der in Abschnitt 6.2 dargestellten Typmanagement-Komponente erbracht werden kann.

D.1.3 Beziehungen zwischen Typen

D.1.3.1 Typvergleich

Anhand der Komponententypen *Savings Bank* und *Bank* kann an dieser Stelle ein exemplarischer Typvergleich illustriert werden, indem Beziehungen zwischen den entsprechenden Typbäumen von der Typmanagement-Komponente evaluiert werden. Für den als potentiellen Substitutionskandidaten festgelegten Komponententyp *Savings Bank* kann zudem eine Aussage über dessen Konformität zur *Bank*-Komponente abgeleitet werden.

Auf der semantischen Ebene konsumiert der Komponententyp *Savings Bank* – wie auch die *Bank*-Komponente – die Business Events *Register*, *Unregister* sowie *TransferRequest*, wobei zudem ein gleiches Protokoll über diese Teilmenge der Business Events auf Seiten der *Savings Bank* vorhanden ist. Für die zusätzlich definierten Business Events *AccountInfo* und *BuildingSavingsAgreement* wird dabei ein zweites Protokoll von der *Savings Bank* bereitgestellt. Letztendlich werden von beiden Komponententypen keine Constraints definiert, was in einer `equal`-Assoziation (Gleichheit) zwischen den Constraint-Mengen resultiert, wobei sich für die Protokollmenge sowie die Menge der emittierten beziehungsweise konsumierten Business Events für die *Savings Bank* jeweils eine `extension`-Assoziation (Erweiterung) ableiten lässt. Somit stellt der Komponententyp *Savings Bank* eine Typart-1 (Subtyp) des Basistyps *Bank* dar, sodass dieser dadurch eine zur *Bank* (semantisch) konforme Typausprägung darstellt.

Der exemplarische Vergleich der syntaktischen Realisierungen des Business Events *TransferRequest* auf der Nachrichtenebene zeigt dann, dass die *Savings Bank* Subtyp-Entsprechungen für die semantischen Nachrichtentypen *prüfeKunde* und *überweise* der *Bank* (`subtype`-Assoziation) sowie einen weiteren Nachrichtentyp *prüfeLiquidität* spezifiziert. Die Protokollmenge auf der Seite der *Savings Bank* zeichnet sich dabei durch ein zur *Bank* identisches sowie ein weiteres Protokoll über die entsprechenden Nachrichten aus. Folglich stellt die Menge der semantischen Nachrichtentypen wie auch deren Protokollmenge auf Seiten der *Savings Bank* eine Erweiterung `extension` im Vergleich

zur *Bank* dar, sodass der syntaktische Business-Event-Typ der *Savings Bank* als Typart-1 (Subtyp) der *Bank* eingestuft wird. Da es sich in diesem Fall um ein konsumiertes Business Event handelt, sind für diese Beziehung die subtypisierten semantischen Nachrichtentypen auf Seiten der *Savings Bank* verantwortlich, sodass die *Savings Bank*-Komponente im Hinblick auf die syntaktische Realisierung des Business Events *TransferRequest* zur *Bank* konform ist.

Allerdings unterscheiden sich die Protokolle beider Komponententypen auf der syntaktischen Ebene in der syntaktischen Realisierung des Nachrichtentyps *überweise*, sodass zwischen beiden Protokollmengen keine Relation besteht. Für die Operationstypen *debit* und *deposit* ist auf Seiten der *Savings Bank* eine Subtyp-Entsprechung (`subtype`-Assoziation) sowie ein zusätzlicher Operationstyp *checkPIN* vorhanden, wobei zwischen den Attributen Kommunikationstyp sowie Kommunikationsmodelltyp jeweils deren Gleichheit durch die `equal`-Assoziation zum Ausdruck gebracht wird. Für den syntaktischen Nachrichtentyp der *Savings Bank* wird demnach die Beziehung einer Typart-2 einer Typart-1 abgeleitet, sodass die (syntaktische) Konformität der *Savings Bank* zur *Bank* in diesem Punkt nicht gegeben ist.

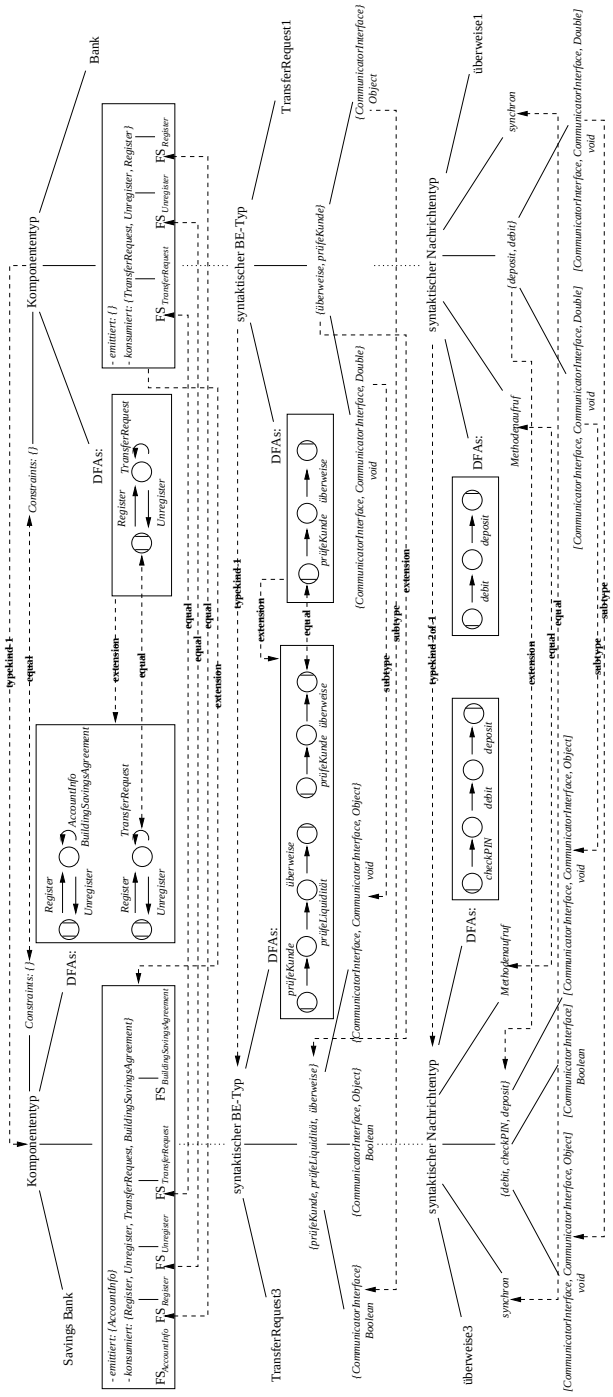
D.1.3.2 Kompatibilitätsprüfung

Die Komponententypen *Customer* und *Bank* können im Folgenden exemplarisch einer Kompatibilitätsprüfung unterzogen werden, indem in diesem Fall Kompatibilitätsbeziehungen zwischen den entsprechenden Typbäumen von der Typmanagement-Komponente abgeleitet werden.

Beide Komponententypen *Customer* und *Bank* basieren auf der semantischen Ebene auf einem gleichen, gemeinsamen Protokoll, wobei die darin definierten Business Events vom *Customer* emittiert sowie von der *Bank* konsumiert werden können. Auf der Basis dieses inversen Interaktionsmusters beider Komponententypen hinsichtlich des zu Grunde liegenden Protokolls ergibt sich schließlich die (semantische) Kompatibilität von *Bank* und *Customer*, ausgedrückt durch die Assoziation `compatibleType`. Irrelevant sind an dieser Stelle die leeren Constraints, da sie keine logischen Randbedingungen enthalten, die im Kontext einer Kompatibilitätsprüfung ausgewertet werden müssten.

Auf der Nachrichtenebene materialisiert sich das Business Event *TransferRequest* der Interaktionspartner in verschiedenen syntaktischen Realisierungen. Dabei unterscheiden sich die Mengen semantischer Nachrichtentypen sowie die Parameter-

Abbildung D.7
Assoziationen zwischen
Savings Bank *und* Bank



typen einander entsprechender Nachrichtentypen. Dennoch ergibt sich eine Kompatibilitätsbeziehung `compatibleType` zwischen den syntaktischen Realisierungen des Business Events der *Bank* und des *Customer* auf Basis des gemeinsamen Protokolls beider Komponententypen, wobei bezüglich der an den Zustandsübergängen angegebenen semantischen Nachrichtentypen deren gegenseitige Kompatibilität gewährleistet ist, ausgedrückt durch `compatibleType`-Assoziationen. Die Kompatibilitätsbeziehung stützt sich dabei auf die subtypisierten semantischen Nachrichtentypen auf Seiten der aufgerufenen, das Business Event konsumierenden Komponente – hier *Bank* – im Vergleich zu denen der aufrufenden, das Business Event emittierenden Komponente – in diesem Fall *Customer*. Genau in diesem Fall entspricht die Benutzung der Funktionalität an der typisierten Komponentenschnittstelle den Regeln, die die angebotene Schnittstelle festlegt.

Analog gilt dies auch auf der syntaktischen Ebene für die syntaktischen Realisierungen des Nachrichtentyps *überweise* von *Customer* und *Bank*. Durch das gemeinsame Protokoll beider Interaktionspartner sowie den einander entsprechenden kompatiblen Operationstypen *debit* beziehungsweise *deposit* ergibt sich somit eine Kompatibilitätsbeziehung `compatibleType`. Zudem basieren beide Komponententypen auf dem gleichen Kommunikationstyp sowie Kommunikationsmodelltyp, ausgedrückt durch jeweils eine `equal`-Assoziation. Die syntaktischen Nachrichtentypen beider Komponententypen sind folglich ebenso (syntaktisch) kompatibel.

D.1.4 Adapter

Für den Fall einer Typsubstitution der *Bank* durch die *Savings Bank* wird im vorhergehenden Beispiel ein Adapter zur Überbrückung der nicht-konformen syntaktischen Realisierung des Nachrichtentyps *überweise* der *Savings Bank* benötigt. Diese Adaption ist dabei obligatorisch, um eine typsichere (korrekte) Interaktion von *Customer* und *Savings Bank* sicherzustellen. Mit Ausnahme dieser Inkompatibilitätsüberbrückung hat der Adapter sonst ausschließlich die Aufgabe der Weiterleitung der Domänenaktivitäten.

Um einen beidseitig-vollständig kompatiblen Adapter zu realisieren, der zwischen die *Customer*- und die *Savings Bank*-Komponente geschaltet werden kann, muss auf der semantischen Ebene für beide Komponententypen jeweils ein Protokoll ausgewählt werden, das der nachfolgenden Interaktion zu Grun-

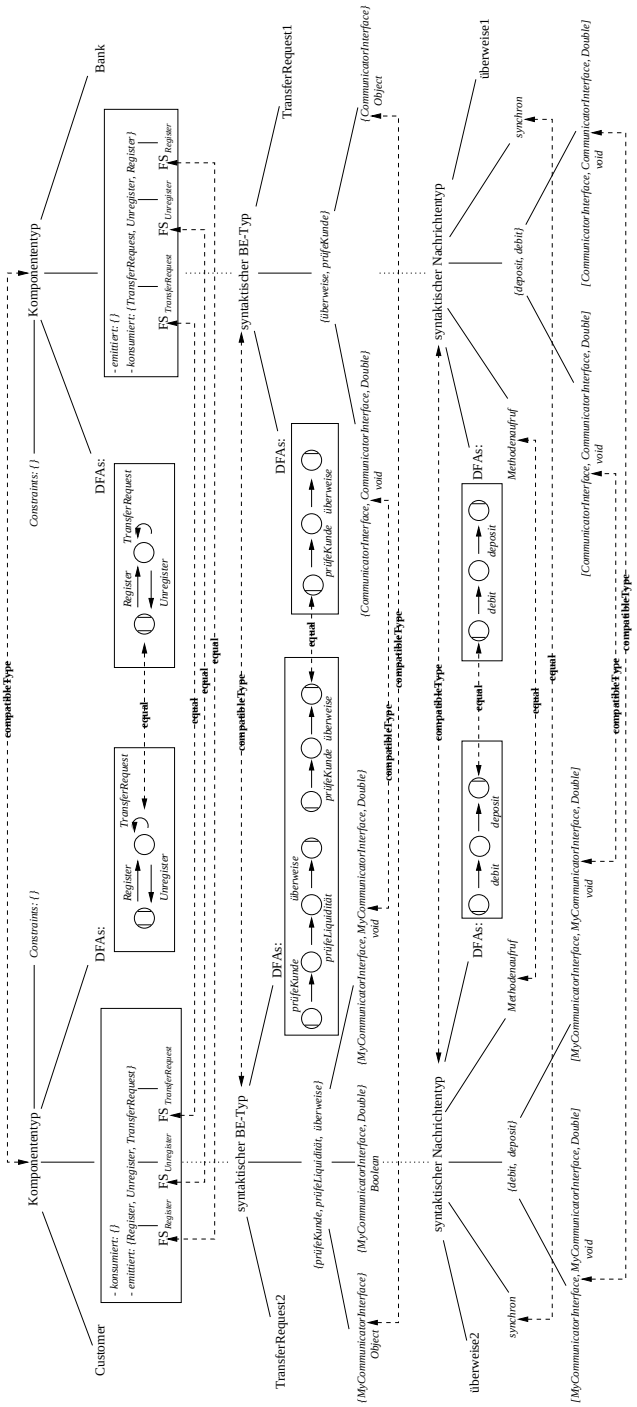


Abbildung D.8
Assoziationen zwischen Customer *und* Bank

de liegen soll. Die beiden ausgewählten Protokolle, deren Interaktionsmuster dann invertiert werden müssen, repräsentieren letztendlich die beiden Adapterschnittstellen auf semantischer Ebene. Im einfachsten Fall kann für die Adapterspezifikation auf der Nachrichtenebene sowie der syntaktischen Ebene die jeweilige syntaktische Repräsentation der *Customer*- (*TransferRequest2*, *überweise2*) beziehungsweise der *Savings Bank*-Komponente (*TransferRequest3*, *überweise3*) redefiniert werden.

D.1.5 Polymorphismus

Schließlich lassen sich die in Abschnitt 4.8 diskutierten verschiedenen Arten von Polymorphismus exemplarisch anhand der Komponententypen *Bank*, *Savings Bank* und *Customer* aufzeigen.

Die *Savings Bank* wurde im gegebenen Beispiel als semantische Typart-1 (Subtyp) des Komponententyps *Bank* dargestellt, sodass sie eine polymorphe Komponente darstellt, denn sie kann einerseits in der Rolle der *Savings Bank* auftreten, andererseits aber auch in der *Bank*-Rolle, dem Kontext des Basistyps. Bezüglich der syntaktischen Realisierung des Nachrichtentyps *überweise* bestand dabei allerdings keine Konformität (s.o.), sodass – wie bereits angesprochen – ein Adapter benötigt wird, der die Inkompatibilität bezüglich dieses syntaktischen Nachrichtentyps überbrückt.

Des Weiteren kann der Fall der expliziten Typumwandlung (Coercion) betrachtet werden, der exemplarisch an der *Bank*-Komponente demonstriert werden soll, wenn diese die Rolle des *Customer* einnimmt. Praktisch würde dies etwa dann auftreten, wenn die *Bank* beispielsweise gegenüber einem Komponententyp *Federal Bank* (Bundesbank) in der *Customer*-Rolle auftritt. Folglich stellt die *Bank* dann nicht mehr die bedienende Komponente (Server) dar, sondern vielmehr diejenige, die die angebotenen Dienstleistungen in Anspruch nimmt (Client). Allerdings sollte für die Komponente ein Default-Typ – in diesem Fall *Bank* – zu Grunde gelegt werden, der die standardmäßige Verwendung der Komponente beschreibt.

Typkonvertierbare Komponenten werden als Komponenten mit mehr als einer semantischen Ebene modelliert. Bei der nachfolgenden Typregistrierung werden diese Semantikrepräsentationen mit der Nachrichtenebene der Komponente verknüpft, die damit bezüglich ihrer Semantik mehrdeutig ist. Die Abbildung D.9 zeigt exemplarisch den syntaktischen Business-Event-Typ *TransferRequest1*, der von zwei semantischen Ebenen – hier

Bank und *Customer* – zugänglich ist. Die als Default-Typ ausgezeichnete *Bank*-Repräsentation verweist schließlich auf alle weiteren Rollen, die die Komponente einnehmen kann, wobei jede Rolle wiederum auf ihren Default-Typ verweist.

D.2 Druckdienst-Beispiel

Ein zusätzliches Anwendungsbeispiel soll im Folgenden weitere Aspekte möglicher Typbeziehungen verdeutlichen und ist durch einen Komponententyp *Postscript Print Service* (Druckdienst) gegeben, zu dem zwei Substitutionskandidaten *Print Service* sowie *TEX Print Service*[1] definiert werden. Der *Postscript Print Service* kann dabei lediglich Postscript-Druckaufträge verarbeiten, während der *Print Service* die Verarbeitung beliebiger Druckaufträge gestattet. Der *TEX Print Service* lässt sich dann als Druckdienst klassifizieren, der neben Postscript-Dokumenten auch PDF-Dokumente verarbeiten kann. Als potenzielle Interaktionspartner stehen zwei Komponententypen *Address Service* sowie *Info Service* zur Verfügung.

Im Folgenden werden zunächst wiederum die Typbeschreibungen der Druckdienste sowie der Interaktionspartner festgelegt, wobei abermals aus Platzgründen lediglich ein Querschnitt durch den zu Grunde liegenden Typbaum dargestellt wird. Dabei liegt der Fokus auf der Darstellung der Beschreibung des Druckauftrags auf den drei Ebenen des Typbaums.

D.2.1 Typbeschreibungen der Dienste

Die Abbildung D.10 zeigt einen Auszug des Typbaums des Komponententyps *Postscript Print Service*.

Der Komponententyp *Postscript Print Service* wird auf der semantischen Ebene durch das Konsumieren des Business Events *PSPrint* (Druckauftrag) sowie das Emittieren des Business Events *PSPrintInfo* (Druckbericht) charakterisiert, wobei deren Semantik – wie auch das auf dieser Ebene definierte Constraint, das die Existenz einer Warteschlage beschreibt – durch Feature Structures repräsentiert wird. Ein Protokoll über Business Events spezifiziert letztendlich alle gültigen Sequenzen von Business Events in Form eines deterministischen endlichen Automats.

[1] In diesem Szenario wird davon ausgegangen, dass ein TEX-Dokument entweder als Postscript- oder als PDF-Dokument ausgegeben wird.

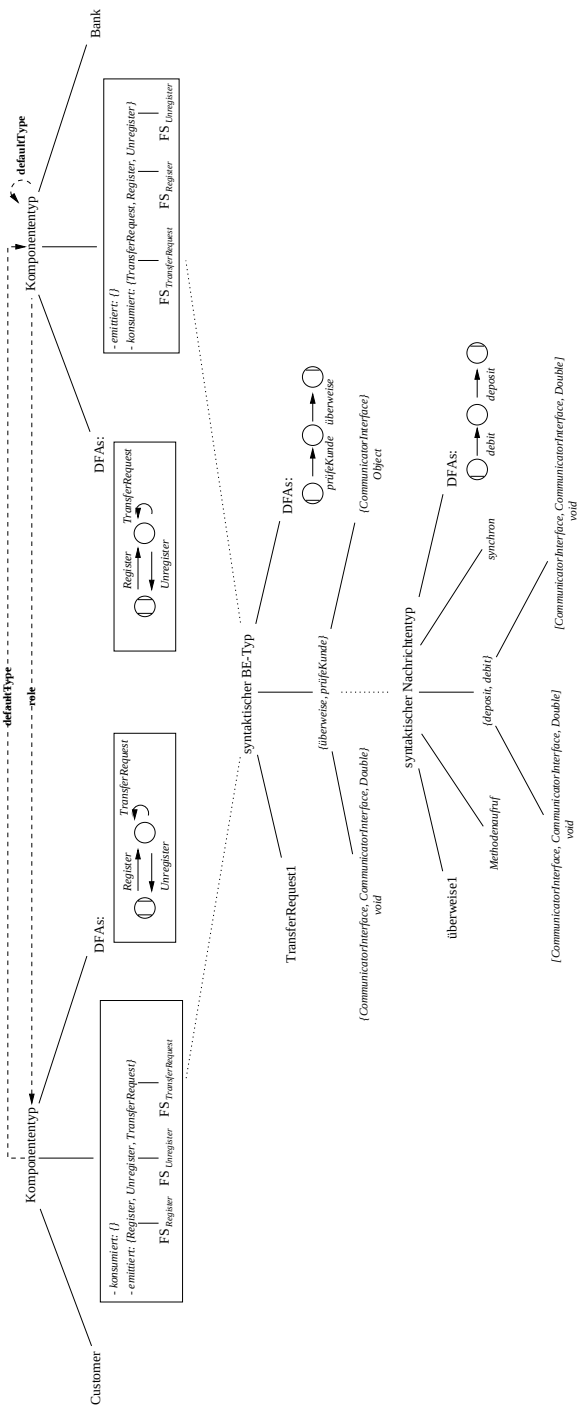


Abbildung D.9
typkonvertierbare (polymorphe) Bank-*Komponente*

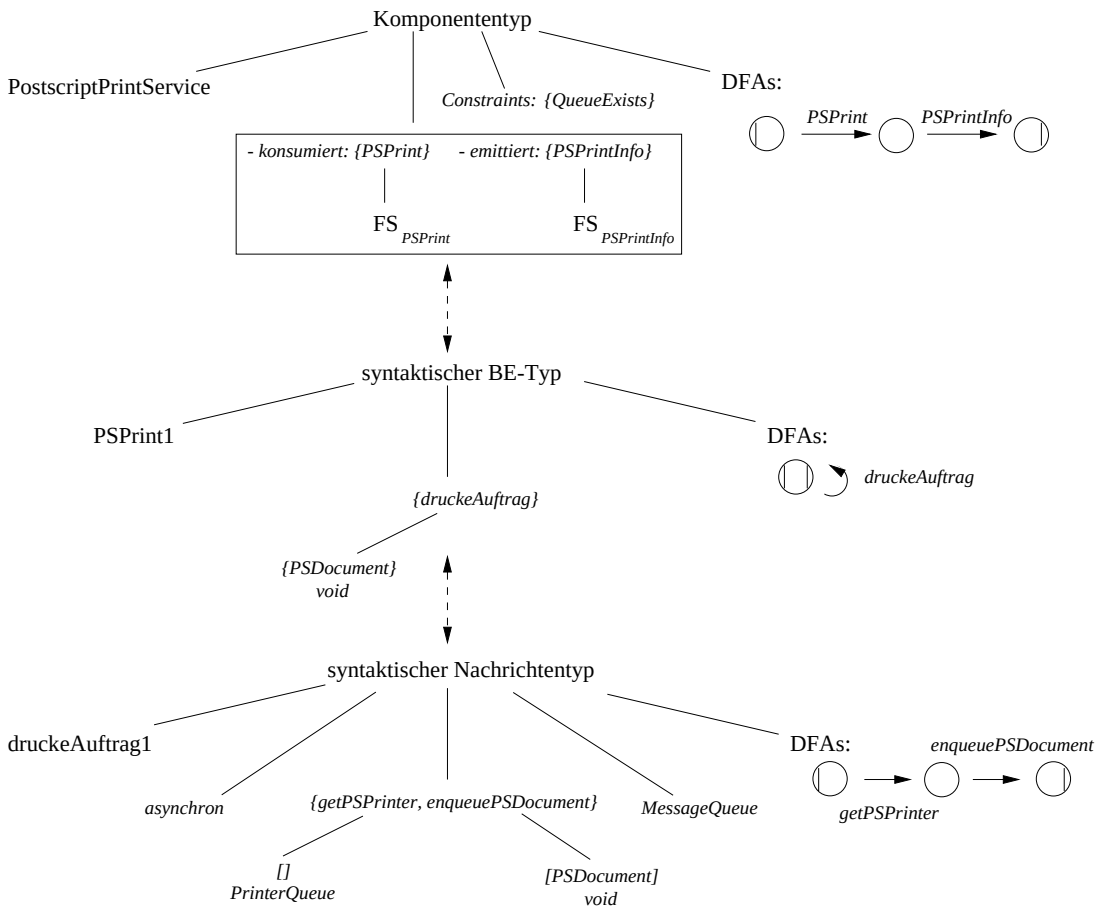


Abbildung D.10
Komponententyp
Postscript Print Service

Das Business Event *PSPrint* materialisiert sich auf der Nachrichtenebene in einem semantischen Nachrichtentyp *druckeAuftrag*, dessen Protokoll wie auch die Datentypmenge der Parameter – hier lediglich `PSDocument` – inklusive eines deklarierten Rückgabetyps – in diesem Fall `void` – festgelegt werden.

Auf der syntaktischen Ebene konkretisiert sich dieser in den Operationstypen (Signaturen) *getPrinter* sowie *enqueuePSDocument*. Das Protokoll über Operationsaufrufe wie auch die Attributwerte des Kommunikationstyps – hier *asynchron* – und des Kommunikationsmodelltyps – in diesem Fall *MessageQueue* – stellen weitere Metainformationen auf dieser Ebene dar.

D.2.1.1 Substitutionskandidaten

In Abbildung D.11 ist ein Ausschnitt des Typbaums des *Print Service* dargestellt, der einen Substitutionskandidaten des *Postscript Print Service* darstellt.

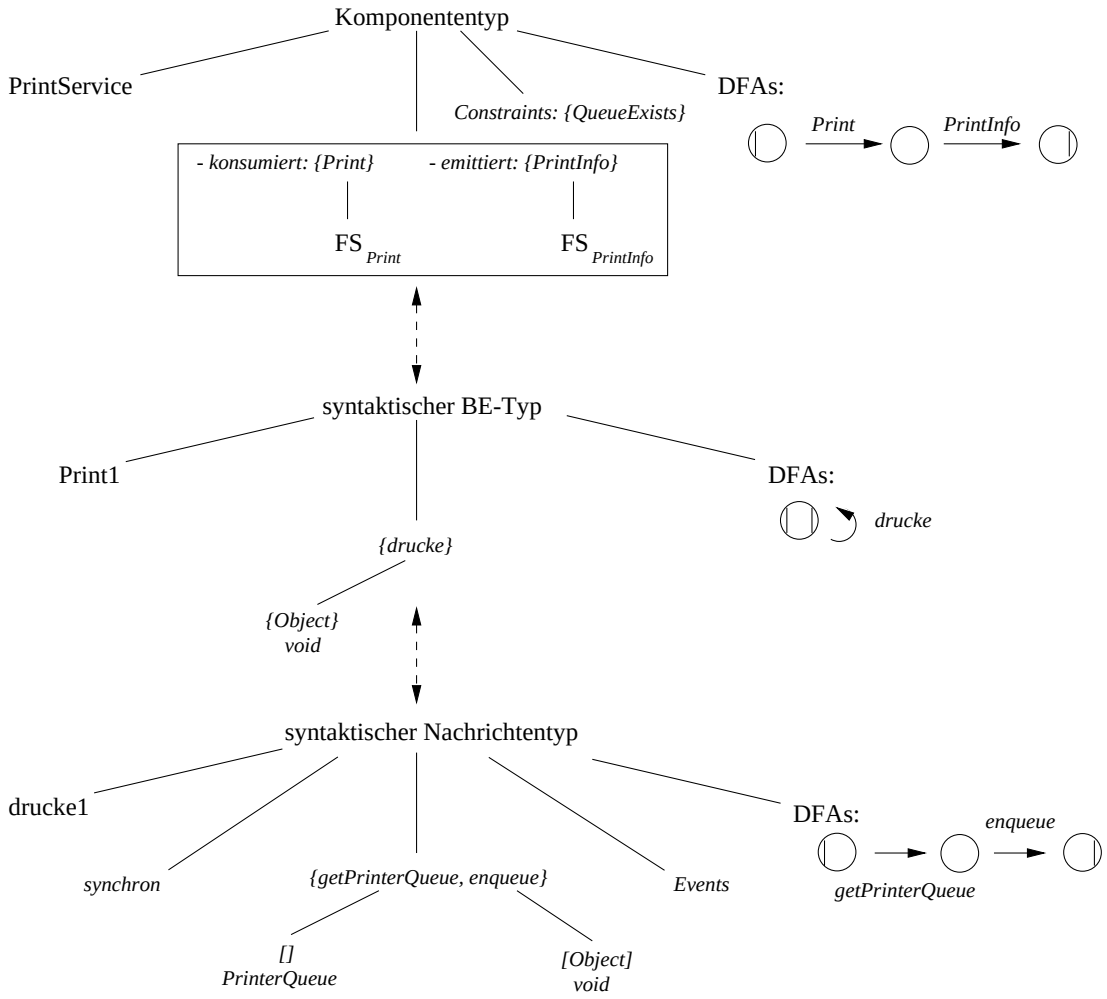


Abbildung D.11 *Komponententyp* Print Service

Semantisch lässt sich der Komponententyp *Print Service* durch das Konsumieren des Business Events *Print* (Druckauftrag) sowie das Emittieren des Business Events *PrintInfo* (Druckbericht) beschreiben, wobei ein zusätzliches Constraint wiederum die Existenz einer Warteschlange angibt. Die Semantik der Business Events sowie das zu Grunde gelegte Constraint werden abermals durch Merkmalsstrukturen repräsentiert. Somit trägt der *Print Service* dem Interaktionspotential des *Postscript Print*

Service Rechnung, wobei die Verwendung der abstrakten Schnittstelle des *Print Service* in Form eines deterministischen endlichen Automats spezifiziert wird, der das Protokoll des Komponententyps auf semantischer Ebene ausmacht.

Die exemplarisch angegebene syntaktische Realisierung des Business Events *Print* umfasst auf der Nachrichtenebene lediglich einen semantischen Nachrichtentyp *drucke*, dessen Protokoll wie auch die Datentypmenge der Parameter – in diesem Fall `Object` – inklusive des Rückgabetyps `void` festgelegt sind. Es bleibt dabei festzuhalten, dass der *Print Service* somit die Generalisierung der vom *Postscript Print Service* angebotenen Funktionalität realisiert.

Auf der syntaktischen Ebene drückt sich die syntaktische Realisierung des Nachrichtentyps in den Operationstypen *getPrinterQueue* und *enqueue* aus, wobei wiederum ein Protokoll die Sequenzen potentieller Operationsaufrufe spezifiziert. Die Attributwerte des Kommunikationstyps sowie des Kommunikationsmodelltyps sind auf *synchron* beziehungsweise *Events* festgelegt.

Auf der Grundlage der Typbeschreibung des *Postscript Print Service* kann eine weitere Typausprägung *TEX Print Service* definiert werden, indem die entsprechende Typbeschreibung um ein weiteres Segment erweitert wird, das die Verarbeitung von PDF-Druckaufträgen definiert. Während die durch den *Postscript Print Service* gegebene Typbeschreibung vom *TEX Print Service* redefiniert wird, sodass in diesem Punkt beide Typbeschreibungen übereinstimmen, hier sogar identisch sind, stellt die Neudefinition des Segments zur Verarbeitung von PDF-Druckaufträgen eine Erweiterung des *TEX Print Service* um eine Zusatzfunktionalität dar – ganz im Gegensatz zur Generalisierung der Druck-Funktionalität des *Postscript Print Service* durch den *Print Service*.

Beide Segmente des *TEX Print Service* unterscheiden sich dabei lediglich auf der Nachrichtenebene sowie der syntaktischen Ebene, sodass ihnen gleiche Semantikrepräsentationen zu Grunde liegen. Auf eine Ansicht der Typbeschreibung des *TEX Print Service*, der somit einen weiteren Substitutionskandidaten (Typart-1) darstellt, soll allerdings an dieser Stelle verzichtet werden.

Zu den beschriebenen Druckdiensten sollen im Folgenden weitere Komponententypen definiert werden, die als potentielle Interaktionspartner fungieren. Dies können beispielsweise

Komponententypen sein, die die beschriebenen Druckdienste verwenden. Durch die Definition möglicher Interaktionspartner lässt sich dann eine exemplarische Kompatibilitätsprüfung durchführen, während die potenziellen Substitutionskandidaten im Rahmen eines Typvergleichs auf Konformität im Hinblick auf den zu ersetzenden Komponententyp *Postscript Print Service* analysiert werden können.

D.2.1.2 Interaktionspartner

Als potentieller Interaktionspartner der Druckdienste steht ein Komponententyp *Address Service* zur Verfügung, der Postscript-Druckaufträge generiert.

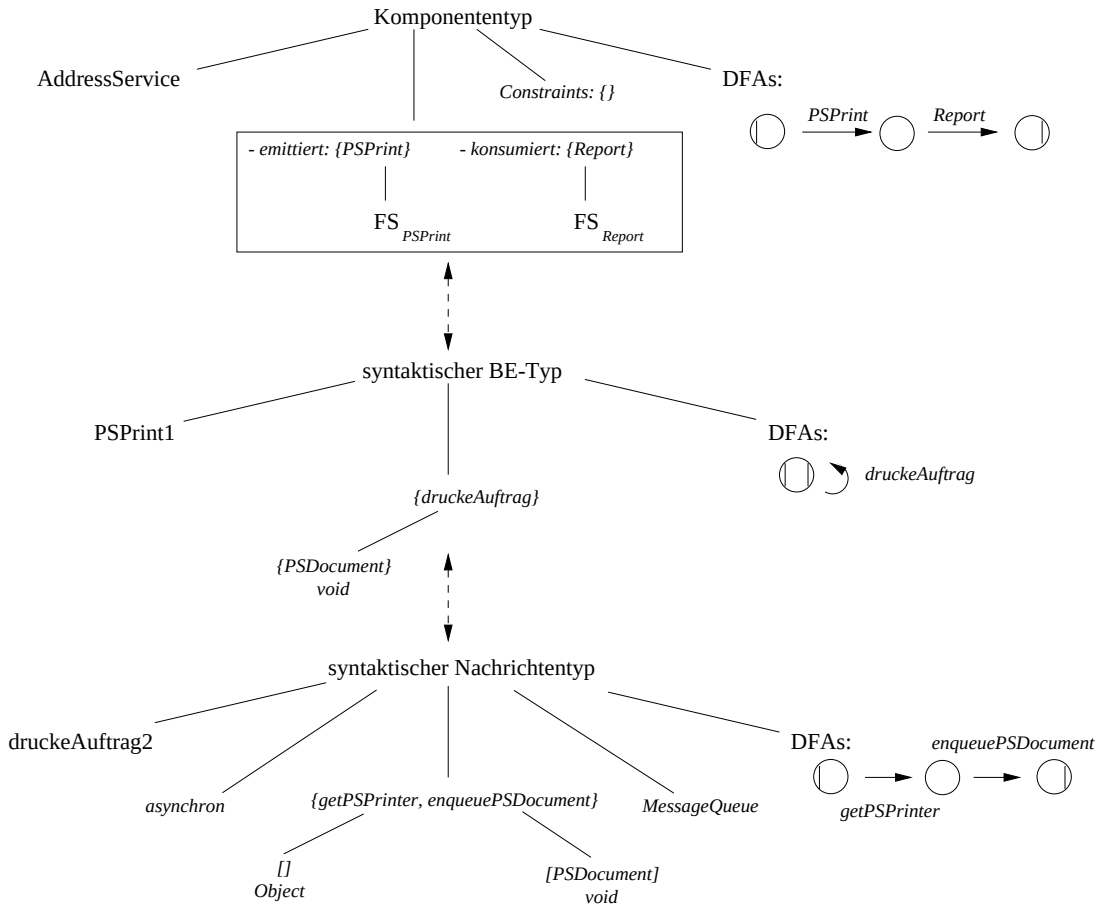


Abbildung D.12
Komponententyp
Address Service

Als potentieller Interaktionspartner zeichnet sich der *Address Service* auf der semantischen Ebene durch das Emittieren

des Business Events *PSPrint* (Druckauftrag) sowie das Konsumieren des Business Events *Report* (Druckbericht) aus, wobei sich deren Semantikrepräsentation wiederum auf Feature Structures stützt. Der Komponententyp setzt dabei die gleiche Reihenfolge der Domänenaktivitäten wie die Druckdienste voraus, ausgedrückt durch das spezifizierte Protokoll über Business Events. Hinsichtlich dieses Protokolls liegt demnach auf Seiten des *Address Service* ein inverses Interaktionsmuster vor, wobei dieser keine typrelevanten Regeln in Form von Constraints vorsieht.

Auf der Nachrichtenebene wie auch auf der syntaktischen Ebene basiert der vorliegende Auszug des Typbaums der syntaktischen Realisierung des Druckauftrags *PSPrint* – mit Ausnahme der Signatur von *getPSPrinter* – auf dem Typbaum des *Postscript Print Service*.

Schließlich soll nun ein Komponententyp *Info Service* definiert werden, der Druckaufträge in ASCII-Form generiert. Durch diese Eigenschaft steht bereits an dieser Stelle die syntaktische Inkompatibilität zu den Komponententypen *Postscript Print Service* sowie *TEX Print Service* fest, sodass auf eine Angabe der Typbeschreibung verzichtet werden soll. Allerdings ergibt sich eine Kompatibilitätsbeziehung zum *Print Service*, da dieser die Verarbeitung beliebiger Druckaufträge erlaubt.

Vor der Evaluierung von Beziehungen zwischen den exemplarisch definierten Komponententypen soll zunächst noch auf die Semantikrepräsentation der Business Events eingegangen werden.

D.2.2 Repräsentation der Semantik der Business Events

Für die Business Events der Komponententypen des Beispielszenarios sollen wiederum die jeweiligen Semantikrepräsentationen vorgestellt werden. Die Semantik der Business Events wird dabei wiederum durch Merkmalsdiagramme repräsentiert, die sich in diesem Fall auf die aus der Domänenanalyse hervorgegangenen standardisierten Merkmale der Typdomäne *Druckdienste* stützen.

Die Business Events *Print* sowie *PSPrint* basieren auf dem in Abbildung D.13 gezeigten Merkmalsdiagramm, das die Semantik eines *Druckauftrags* darstellt.

Das Merkmalsdiagramm des *Druckauftrags* wird durch ein Merkmal *PRINT ORDER* (Druckauftrag) charakterisiert, das sich aus drei elementaren Submerkmalen *PRINTER* (Drucker)

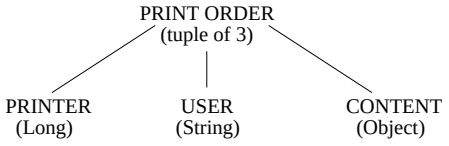


Abbildung D.13 *Merkmalsdiagramm eines* Druckauftrags

des Typs `Long`, *USER* (Benutzer) des Typs `String` sowie *CONTENT* (Inhalt) des Typs `Object` zusammensetzt.

Für die Business Events *PrintInfo*, *PSPrintInfo* sowie *Report* wurde das in Abbildung D.14 gezeigte Merkmalsdiagramm zu Grunde gelegt. Es repräsentiert die Semantik eines *Druckberichts*.

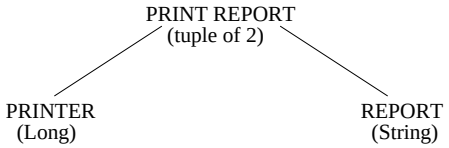


Abbildung D.14 *Merkmalsdiagramm eines* Druckberichts

Das Merkmalsdiagramm des *Druckberichts* zeichnet sich durch ein zusammengesetztes Merkmal *PRINT REPORT* (Druckbericht) aus, das zwei elementare Submerkmale *PRINTER* (Drucker) des Typs `Long` sowie *REPORT* (Bericht) des Typs `String` subsummiert.

Im Folgenden können schließlich die Beziehungen zwischen den exemplarisch definierten Komponententypen *Postscript Print Service*, *Print Service* sowie *Address Service* von der in Abschnitt 6.2 vorgestellten Typmanagement-Komponente abgeleitet werden.

D.2.3 Beziehungen zwischen Typen

D.2.3.1 Typvergleich

Auf Basis der zu Grunde liegenden Komponententypen *Postscript Print Service* und *Print Service* soll nachfolgend ein exemplarischer Typvergleich vorgenommen werden, indem Beziehungen zwischen den korrespondierenden Typbäumen von der Typmanagement-Komponente evaluiert werden.

Auf semantischer Ebene ergibt sich zwischen den Komponententypen *Postscript Print Service* und *Print Service* eine `equal-`

Assoziation, die die (semantische) Gleichheit beider Komponententypen reflektiert. Diese ergibt sich auf Grund des identischen Interaktionsmusters – ausgedrückt über emittierte und konsumierte Business Events – sowie einer gleichen Constraint- beziehungsweise Protokollmenge. Beide Komponententypen sind folglich zueinander (semantisch) konform.

Die Ableitung der Beziehungen zwischen den syntaktischen Realisierungen des Druckauftrags *PSPrint1* und *Print1* resultiert auf der Nachrichtenebene in einer Typart-1 (Subtyp) auf Seiten des *Print Service*, da dessen Nachrichtentyp *drucke* einen Subtyp des Nachrichtentyps *druckeAuftrag* des *Postscript Print Service* beschreibt, ausgedrückt durch die `subtype`-Assoziation. Für die Subtypisierung der syntaktischen Realisierung des Druckauftrags müssen schließlich kovariante semantische Nachrichtentypen sichergestellt werden, da es sich um ein konsumiertes Business Event handelt. Letztendlich liegt beiden Komponententypen auf dieser Ebene ein gleiches, gemeinsames Protokoll zu Grunde (`equal`-Assoziation), sodass der *Print Service* bezüglich dieses syntaktischen Business-Event-Typs eine konforme Typausprägung zum *Postscript Print Service* darstellt.

Im Hinblick auf die Realisierungen des Druckauftrags auf der syntaktischen Ebene ergibt sich zwischen den korrespondierenden syntaktischen Nachrichtentypen *druckeAuftrag1* und *drucke1* die Beziehung einer Typart-3 (technischer Typ) einer Typart-1 (Subtyp) auf Seiten des *Print Service*. Verantwortlich dafür sind in diesem Fall die Differenzen beider Komponententypen in Bezug auf den Kommunikations- sowie Kommunikationsmodelltyp wie auch die resultierenden Relationen zwischen einander entsprechenden Operationstypen, für die an dieser Stelle ein gleiches, gemeinsames Protokoll vorliegt (`equal`-Assoziation). Die Operationstypen des Substitutionskandidaten *Print Service* stellen dabei typgleiche beziehungsweise Subtyp-Entsprechungen des *Postscript Print Service* dar, ausgedrückt durch die `equal`- beziehungsweise `subtype`-Assoziation. Folglich ist der *Print Service* ebenso syntaktisch zum *Postscript Print Service* konform, wobei die obligatorische Integration eines Standardadapters vorgenommen werden muss, der die Adaption des Kommunikationstyps sowie des Kommunikationsmodelltyps realisiert. Diese Zuordnung wird automatisch von der Typmanagement-Komponente durchgeführt.

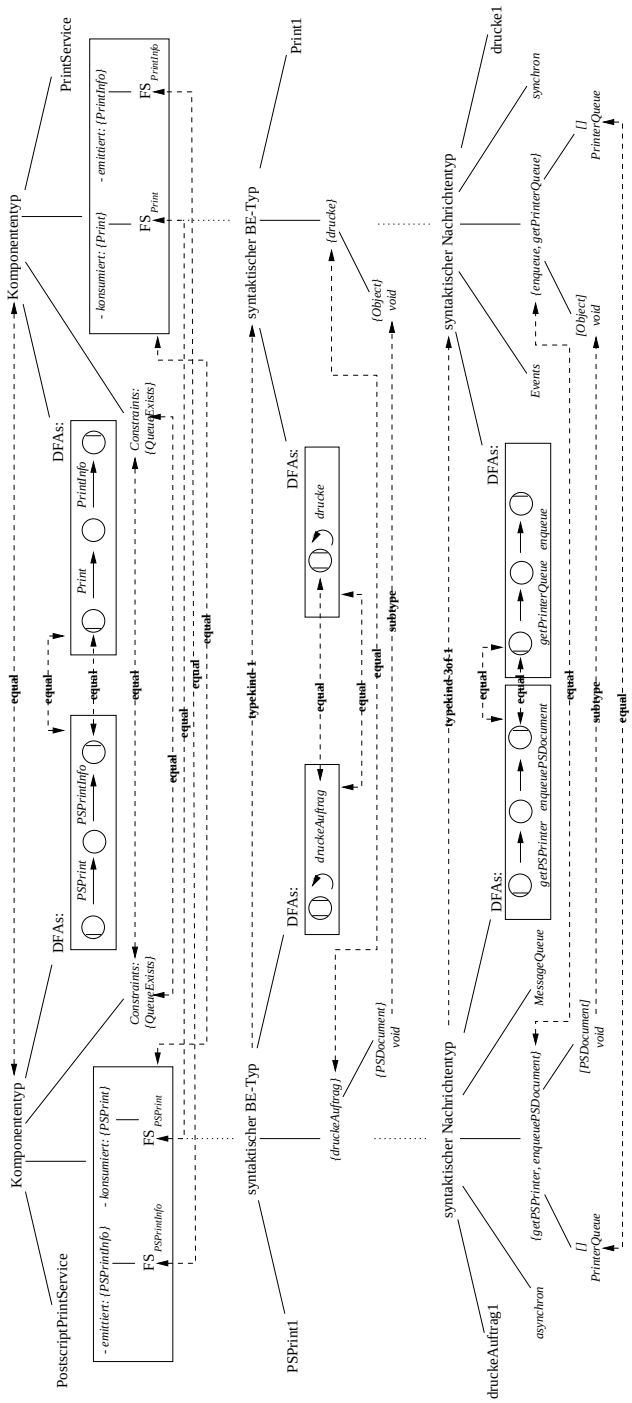


Abbildung D.15
Assoziationen zwischen Postscript Print Service *und* Print Service

D.2.3.2 Kompatibilitätsprüfung

Die Komponententypen *Address Service* und *Postscript Print Service* können exemplarisch auf gegenseitige Kompatibilität geprüft werden, indem in diesem Fall Kompatibilitätsbeziehungen zwischen den Typbäumen beider Komponenten vom Typmanager evaluiert werden.

Auf der semantischen Ebene lässt sich zwischen den Komponententypen *Address Service* und *Postscript Print Service* eine Kompatibilitätsbeziehung `compatibleType` ableiten, die sich auf Basis des inversen Interaktionsmusters beider Interaktionspartner hinsichtlich des spezifizierten Protokolls ergibt. Der durch den *Address Service* emittierte Druckauftrag kann dabei vom *Postscript Print Service* konsumiert werden, während von diesem ein Druckbericht ausgeht, der wiederum eine eingehende Interaktion des *Address Service* darstellt. Grundlage für die (semantische) Kompatibilität ist schließlich das gleiche, gemeinsame Protokoll beider Interaktionspartner.

Auf der Nachrichtenebene wie auch auf der syntaktischen Ebene lassen sich jeweils Kompatibilitätsbeziehungen auf Grund typgleicher Nachrichten- beziehungsweise Operationstypen sowie gleicher, gemeinsamer Protokolle ableiten. Typgleichheit impliziert dabei stets die Kompatibilität der Typausprägungen. Die einzige Ausnahme stellen in diesem Fall die Signaturen des Operationstyps *getPSPrinter* dar, die zwar voneinander abweichen, aber dennoch kompatibel zueinander sind.

Für den *Address Service* lassen sich nun ebenso Kompatibilitätsbeziehungen zu den anderen beiden Druckdiensten ableiten. Im Falle des *TEX Print Service* ist eine derartige Relation durch die Redefinition der Typbeschreibung des *Postscript Print Service* gegeben, während sich der *Print Service* auf allen drei Ebenen des Typbaums – als konforme Typausprägung des *Postscript Print Service* – ebenso durch Kompatibilität zum *Address Service* auszeichnet.

D.2.4 Adapter

Abgesehen von gegebenenfalls notwendigen Standardadaptern (vgl. a. Abschnitt 4.4.2.1), die automatisch von der Typmanagement-Komponente zugeordnet werden können, ist in diesem Szenario lediglich für eine Kopplung des *InfoService* mit dem *Postscript Print Service* beziehungsweise dem *TEX Print Service* die Notwendigkeit einer Adaption gegeben, da beide Druck-

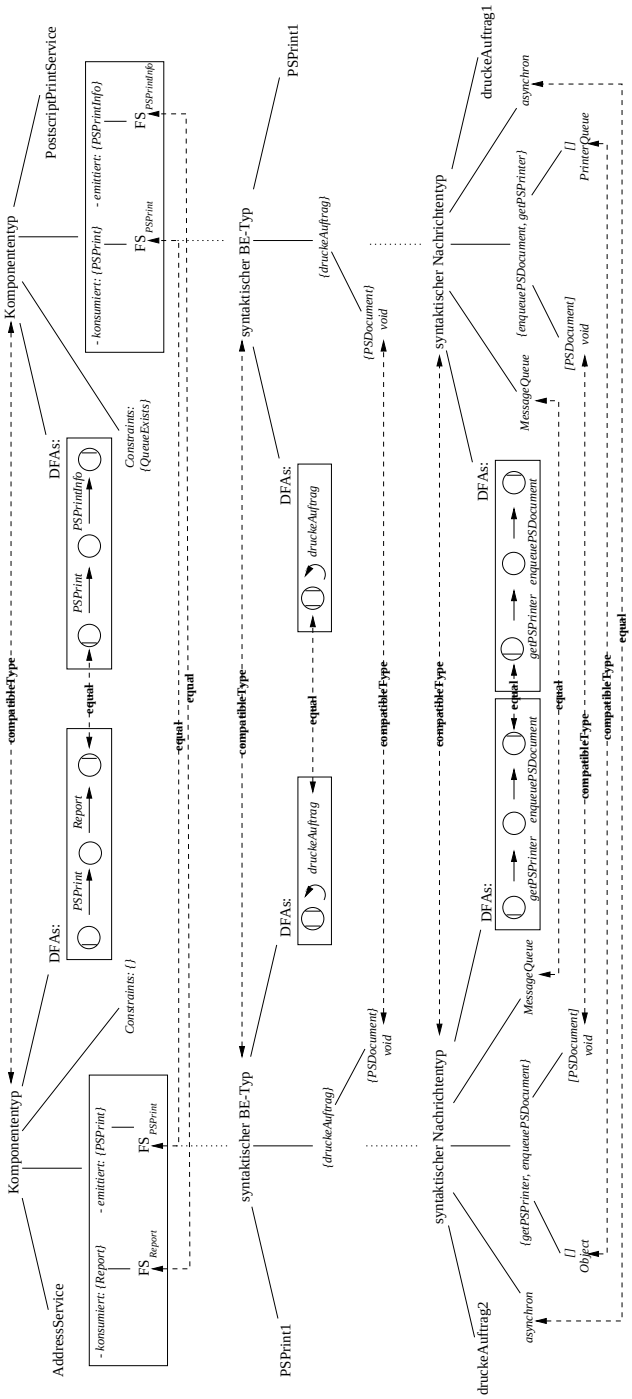


Abbildung D.16
Assoziationen zwischen Address Service *und* Postscript Print Service

dienste keine ASCII-Druckaufträge verarbeiten können. Da der *TEX Print Service* in seiner Postscript-Funktionalität mit dem *Postscript Print Service* identisch ist, wird an dieser Stelle lediglich ein Adaptionsmechanismus zur Überbrückung der Inkompatibilitäten von *InfoService* und *Postscript Print Service* benötigt, der ebenso vom *TEX Print Service* genutzt werden kann. Im Gegensatz dazu ist eine typsichere Interaktion aller Druckdienste mit dem *Address Service* stets sichergestellt.

Für die Realisierung der Adapterkomponente muss dabei wiederum im ersten Schritt für beide zu koppelnde Komponenten jeweils ein Protokoll auf der semantischen Ebene ausgewählt werden, dessen Interaktionsmuster daraufhin invertiert wird. Auf Basis der beiden selektierten Protokolle kann anschließend die Adapterspezifikation vorgenommen werden, indem im einfachsten Fall die den Protokollen zu Grunde liegenden syntaktischen Realisierungen beider Komponenten redefiniert werden.

D.2.5 Polymorphismus

In diesem Beispiel stellt einerseits der *Print Service*, wie auch andererseits der *TEX Print Service* eine polymorphe Komponente dar, da diese bei der Verarbeitung von Postscript-Druckaufträgen jeweils die Rolle des *Postscript Print Service* einnehmen (Subtyp-Polymorphismus).

Des Weiteren lässt sich das Overloading am Beispiel des *TEX Print Service* illustrieren, da dieser die Semantik des Druckauftrags durch zwei verschiedene syntaktische Ausprägungen – den Postscript- sowie den PDF-Druckauftrag – realisiert, die damit den Anwendungskontext (syntaktisch) festlegen. Um die Eindeutigkeit der Semantik-Syntax-Zuordnung zu wahren, müssen in diesem Fall zwei (identische) Semantikrepräsentationen angelegt werden, sodass dies auf der semantischen Ebene zwei (semantisch) gleiche Business Events zur Folge hat, deren syntaktische Ausprägungen erst auf der Nachrichtenebene sowie der syntaktischen Ebene voneinander abweichen.

E Klassenstruktur des Typmodells

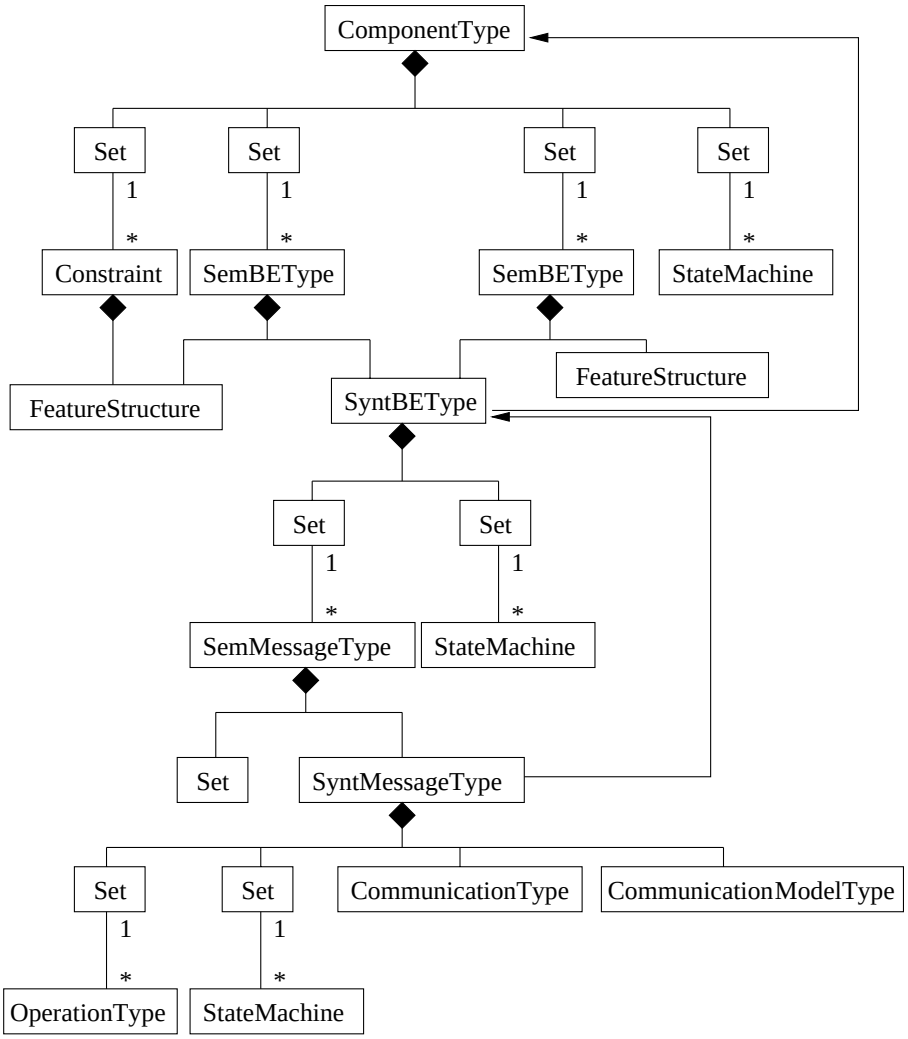


Abbildung E.1
Klassendiagramm des Typmodells

F Omega-Kanten

Im Rahmen des Protokollvergleichs, bei dem die Äquivalenz der den Protokollen zu Grunde liegenden DFAs durch den in Kapitel 3 vorgeschlagenen Algorithmus geprüft wird, soll hier noch auf eine Problematik hingewiesen werden, die in diesem Zusammenhang aus der vorgenommenen Einführung der ω-Kanten (»reserviertes« Wort) resultiert. Äquivalente DFAs akzeptieren dabei die gleiche Sprache und können demnach durch die gleiche reguläre Menge beschrieben werden.

Durch die in Abschnitt 4.3.4 vorgeschlagene Einführung von ω-Kanten als potenzielle Zustandsübergänge kann schließlich die Eindeutigkeit von Protokollen sichergestellt werden, deren Zustandsübergänge strukturell beschrieben sind. In diesem Zusammenhang ist die folgende Diskussion zu betrachten.

Bei der Modellierung – genauer gesagt bei der Festlegung zulässiger Sequenzen von Domänenaktivitäten (Business Events), Nachrichten beziehungsweise Operationsaufrufen – können sich mehrere Möglichkeiten hinsichtlich der zusätzlichen Definition von ω-Kanten ergeben, um nachfolgend die Eindeutigkeit aller Zustandsübergänge zu gewährleisten. Haben zwei Modellierer beispielsweise die gleiche Vorstellung eines Protokolls P mit der Sprache

$L(P) = a^*b$,
mit Supertyp- / Subtyp-Verhältnis zwischen a und b,

weichen aber in ihrer Entscheidung bezüglich der in diesem Fall einzuführenden ω-Kanten voneinander ab, so ist deren Äquivalenz möglicherweise nicht mehr gegeben. Somit gilt $L(P_1) \neq L(P_2)$, für beispielsweise:

$L(P_1) = a^*\omega b$
$L(P_2) = (\omega a^*\omega)^*b$

Die ω-Kanten dürfen dabei nicht als λ-Kanten (»leeres« Wort) interpretiert werden, da sonst nicht mehr von einem deterministischen Automaten gesprochen werden kann. Der Vergleichsalgo-

rithmus würde für dieses Beispiel somit die Ungleichheit der Protokolle P_1 und P_2 ableiten.

Wenn die Bedeutung der ω-Kanten allerdings programmiersprachlich beziehungsweise algorithmisch betrachtet wird, so ist dieses Verhalten geradezu gerechtfertigt. Dies spiegelt sich insbesondere darin wider, dass einer ω-Kante in diesem Zusammenhang vielmehr die Semantik einer (Abbruch-) Bedingung innewohnt, wie sie beispielsweise zur Ablaufsteuerung in sequentiellen Algorithmen verwendet wird. Die Ungleichheit der beiden Protokolle P_1 und P_2 besteht demnach in unterschiedlichen internen Bedingungen für deren Ablauf.

G Assoziationen im Typmodell

Die folgenden tabellarischen Übersichten umfassen eine Gesamtdarstellung der festgelegten Assoziationen im Typmodell, dessen einzelne Ebenen wiederum separat betrachtet werden. Dabei werden die in Abschnitt 4.4.3 vorgestellten Tabellen durch die Assoziationen `supertype`, `reduction` sowie `basetype` vervollständigt, die in Kapitel 4 aus Gründen der Übersichtlichkeit nicht angegeben wurden. Ein Element des Typmodells *a* steht in einer derartigen Beziehung zu einem Element *b*, wenn für das Verhältnis von *b* zu *a* eine `subtype`, `extension` beziehungsweise `typekind`-Assoziation abgeleitet werden kann.

G.1 Semantische Ebene

Die nachfolgende Tabelle G.1 beinhaltet eine vollständige Übersicht der auf der semantischen Ebene festgelegten Assoziationen im Typmodell:

Komponententyp:	`equal`
	`typekind-1, typekind-2`
	`typekind-2of-1`
	`basetype-1, basetype-2`
	`basetype-2of-1`
	`compatibleType, need-adapter-for`

Mengen semantischer BE-Typen:	`equal, extension, reduction`
semantischer BE-Typ:	`equal`
Constraint-Menge:	`equal, extension, reduction`
Constraint:	`equal`
Protokollmenge:	`equal, extension, reduction`
Protokoll:	`equal`

Tabelle G.1
Assoziationen auf semantischer Ebene

G.2 Nachrichtenebene

In Tabelle G.2 ist eine Gesamtübersicht der auf der Nachrichtenebene definierten Assoziationen im Typmodell dargestellt:

syntaktischer BE-Typ:	equal typekind-1, typekind-2 typekind-2of-1 basetype-1, basetype-2 basetype-2of-1 compatibleType, need-adapter-for
Menge sem. Nachrichtentypen:	equal, extension, reduction
semantischer Nachrichtentyp:	equal, subtype, supertype compatibleType
Protokollmenge:	equal, extension, reduction
Protokoll:	equal

Tabelle G.2
Assoziationen auf Nachrichtenebene

G.3 Syntaktische Ebene

Die folgende Tabelle G.3 umfasst eine vollständige Übersicht der auf der syntaktischen Ebene festgelegten Assoziationen im Typmodell:

syntaktischer Nachrichtentyp:	equal, typekind-1, typekind-2, typekind-3, typekind-2of-1, typekind-3of-1, typekind-3of-2, typekind-3of-2of-1 basetype-1, basetype-2, basetype-3, basetype-2of-1, basetype-3of-1, basetype-3of-2, basetype-3of-2of-1 compatibleType, need-adapter-for
Menge von Operationstypen:	equal, extension, reduction
Operationstyp:	equal, subtype, supertype compatibleType
Protokollmenge:	equal, extension, reduction
Protokoll:	equal
Kommunikationstyp:	equal
Kommunikationsmodelltyp:	equal

Tabelle G.3
Assoziationen auf syntaktischer Ebene

H Propositionale Lineare Temporale Logik

Temporale propositionale Logik basiert auf der Aussagenlogik, also auf logischen Formeln über atomaren Aussagen bzw. Propositionen. Dieser Grundansatz wird dann zunächst durch eine zeitabhängige Zuweisung von Wahrheitswerten erweitert. Es werden dabei unendlich viele, diskrete Zeitpunkte auf einer linearen zukunftsorientierten Zeitlinie mit festem Startpunkt betrachtet. Für jeden dieser Zeitpunkte findet eine eigene Interpretation im Sinne der klassischen Aussagenlogik statt.

Als zweite wesentliche Erweiterung werden die bekannten Operatoren der Aussagenlogik um zusätzliche temporale Operatoren ergänzt. Tabelle H.1 gibt diese wieder, dabei seien p bzw. q atomare Propositionen:

Operator	Semantik
$\Box\, p$	“von nun an” p
$\Diamond\, p$	“manchmal” p
$\triangle\, p$	“nächstes mal” p
$p\,\mathcal{U}q$	p “bis” q

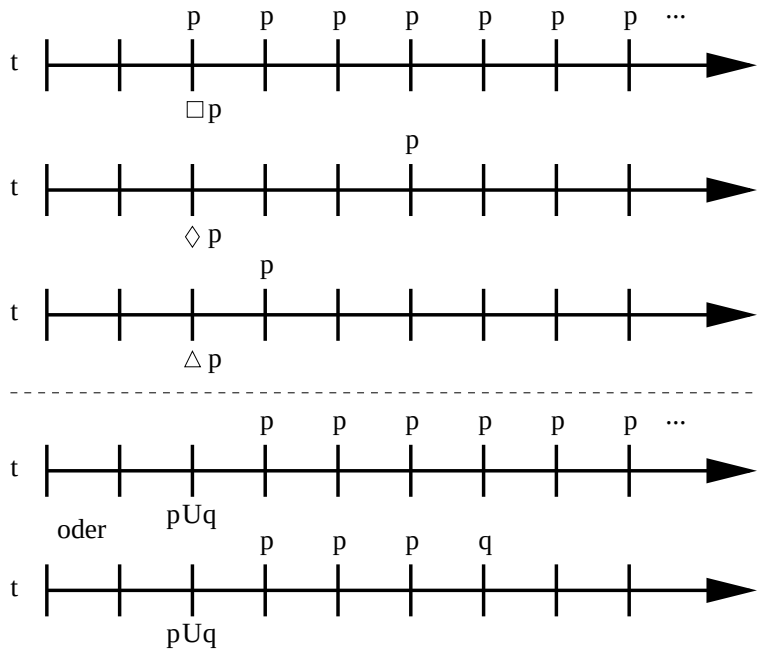
Tabelle H.1
Temporale Operatoren

Die temporalen Operatoren hängen von dem gewünschten logischen System ab und können dementsprechend variieren. Bei der hier getroffenen Auswahl handelt es sich zunächst um eine praktische Grundmenge, die später noch eingehender untersucht wird.

Der von nun an Operator $\Box\, p$ besagt, dass p in diesem und allen folgenden Zeitpunkten den Wert »wahr« besitzt. Der *manchmal* Operator $\Diamond\, p$ sagt aus, dass p möglicherweise in diesem und einigen späteren Zeitpunkten, mit Sicherheit aber irgendwann, den Wert »wahr« annehmen wird. Bei dem *nächstes mal* Operator $\triangle\, p$, ist p genau im folgenden Zeitpunkt »wahr«. Der letzte *bis* Operator $p\,\mathcal{U}q$ besagt, dass entweder p ab jetzt für alle Zeit »wahr« ist, oder genau solange, bis q den Wert »wahr« annimmt.

Die Bedeutung der Operatoren kann am einfachsten mit Hilfe einer Zeitlinie veranschaulicht werden. Zeitpunkte der Linie sind genau dann mit bestimmten Propositionen beschriftet, wenn diese in dem jeweiligen Punkt gelten – also »wahr« sind. Im unteren Bereich wird nun mit dem jeweiligen Operator ein zusammengesetzter Ausdruck über atomaren Propositionen gebildet. Im oberen Bereich wird dann eine mögliche korrespondierende Folge »wahrer« Zeitpunkte dieser atomaren Propositionen angegeben. Abbildung H.1 veranschaulicht die Operatoren auf diese Weise.

Abbildung H.1
Bedeutung von PLTL Operatoren über der Zeitlinie



Im Folgenden werden die soeben informal motivierten Begriffe propositionaler temporaler Logik formal eingeführt.

H.1 Lineare Zeitlinien

Bei der hier betrachteten linearen temporalen Logik ist die Struktur der Zeit eine total geordnete Menge $(S, <)$. Die formale Definition einer solchen Zeitstruktur lautet wie folgt:

Definition 1 (Lineare Zeitstrukturen) *Sei AP eine Menge grundlegender Propositionen. Eine Zeitstruktur ist definiert als*

$M = (S, x, L)$ *mit*

S	*als Menge von Zuständen,*
$x : \mathcal{N} \to S$	*als unendliche Zustandssequenz und*
$L : S \to 2^{AP}$	*als Mengen wahrer Propositionen pro Zustand.*

H.2 PLTL-Syntax

Die PLTL-Syntax besteht aus einem Grundgerüst klassischer Aussagenlogik, ergänzt um die oben eingeführten temporalen Operatoren. Eine entsprechende wohlgeformte PLTL-Formel muss der folgenden Definition genügen:

Definition 2 (PLTL Formeln) *Die Menge gültiger Formeln in propositionaler linearer temporaler Logik (PLTL) ist die kleinste Menge von Formeln, die durch die folgenden Regeln generiert werden:*

(1) Jede atomare Proposition P ist eine Formel.

(2) Sind p und q Formeln, dann sind $(p \wedge q)$ und $\neg p$ Formeln.

(3) Sind p und q Formeln, dann sind $(p \mathcal{U} q)$ und $\triangle\, p$ Formeln.

Als im mathematischen Sinne gebräuchliche Abkürzungen werden des Weiteren die folgenden Operatoren eingeführt:

$$
\begin{aligned}
p \vee q &= \neg(\neg p \wedge \neg q) \\
p \Rightarrow q &= \neg p \vee q \\
p \equiv q &= (p \Rightarrow q) \wedge (q \Rightarrow p) \\
true &= p \vee \neg p \\
false &= \neg true \\
\Diamond\, p &= (true\, \mathcal{U} p) \\
\Box\, p &= \neg \Diamond \neg p \\
\Diamond^{\infty} p &= \Box \Diamond\, p \\
\Box^{\infty} p &= \Diamond \Box\, p \\
(p\, \mathcal{B} q) &= \neg((\neg p)\, \mathcal{U} q)
\end{aligned}
$$

H.3 PLTL-Semantik

Die Semantik von PLTL-Formeln wird in Bezug auf lineare Zeitlinien $M = (S, x, L)$ definiert. Der Ausdruck $M, x \models p$ besagt »In Struktur M ist die Formel p in Zeitlinie x wahr«. $\models$ wird nun induktiv über die Formelstruktur definiert:

(1) $x \models P$ iff $P \in L(s_0)$ für atomare Propositionen P

(2)	$x \models p \wedge q$	iff $x \models p$ und $x \models q$
	$x \models \neg p$	iff Es gilt nicht das $x \models p$
(3)	$x \models (p\,\mathcal{U}q)$	iff $\exists j(x^j \models q$ und $\forall k < j(x^k \models p))$
	$x \models \triangle\, p$	iff $x^1 \models p$

Für die anderen – abgeleiteten – temporalen Operatoren ist eine Definition ihrer Semantik nicht notwendig, kann aber zur besseren Anschauung ebenfalls durchgeführt werden. Für sie gilt bei eigenständiger Definition entsprechend:

$x \models \Diamond\, p$	iff $\exists j(x^j \models p)$
$x \models \Box\, p$	iff $\forall j(x^j \models p)$
$x \models (p\,\mathcal{B}q)$	iff $\forall j(x^j \models q$ impliziert $\exists k < j(x^k \models p))$

H.4 Grundbegriffe des logischen Systems

Um nun das formal eingeführte System anwenden zu können, fehlen noch einige grundsätzliche Definitionen. Im Folgenden wird nach Einführung des *Modellbegriffs* beschrieben, wann PLTL-Formeln *erfüllbar* bzw. *gültig* sein sollen.

Definition 3 (Modell) *Sei p eine PLTL Formel. Jede Zeitstruktur $M = (S, x, L)$ mit $M, x \models p$ definiert ein* ***Modell*** *von p.*

Definition 4 (Erfüllbarkeit) *Eine PLTL Formel p heißt* ***erfüllbar****, falls eine lineare Zeitstruktur $M = (S, x, L)$ existiert, sodass $M, x \models p$*

Definition 5 (Gültigkeit) *Eine PLTL Formel p heißt* ***gültig****, falls jede Zeitstruktur $M = (S, x, L)$ ein Modell von p definiert. Es wird dann $\models p$ geschrieben. Als Implikation ergibt sich, dass p gültig ist, wenn $\neg p$ nicht erfüllbar ist.*

H.5 Verifikation

Um PLTL zur Komponentenspezifikation einsetzen zu können, muss es eine Möglichkeit automatisierter Verifikation entsprechender Beschreibungen geben. Zu diesem Zweck kann die Erfüllbarkeit von PLTL-Formeln durch effiziente Tableau-basierte Algorithmen getestet werden [Manna & Wolper, 1984, Arapis, 1992]. Die Algorithmen erhalten eine Formel F als Eingabe und berechnen einen Graphen (*Erfüllbarkeitgraph*), der alle Modelle repräsentiert, die F erfüllen. Falls F nicht erfüllbar

ist, signalisiert der Algorithmus dies durch einen leere Ausgabe. Da es sich um verbreitet Standardverfahren der Literatur handelt, soll hier auf eine weitere Darstellung verzichtet werden. Den Implementationsarbeiten liegen die Algorithmen aus [Manna & Wolper, 1984] zu Grunde (vgl. a. [Hühnken, 2001]).

I Feature Logik

Feature-Logik[1] ist eine typisierte Form der Logik, die Ausdrücke enthält, die Objektmengen mit bestimmten Merkmalen (engl. *Features*) sowie deren Beziehungen beschreiben. Die Logik kann insbesondere dazu verwendet werden, Objekte mit bestimmten Eigenschaften dynamisch zu selektieren.

I.1 Features und Feature-Terme

Grundelemente der Logik sind *Features*, die die Abstraktion einer bestimmten Eigenschaft repräsentieren. Solche Features werden durch Paare von Namen und zugehörigen Werten charakterisiert, wobei der Bezeichner für die Art einer Eigenschaft steht und der Wert für dessen konkrete Ausprägung. Die konkreten Eigenschaften stehen in Beziehung zu einer Menge von Objekten, denen eben diese Eigenschaften potenziell innewohnen.

Mengen dieser Objekte werden durch *Feature-Terme* beschrieben. Diese Feature-Terme sind logische Ausdrücke über Features. Die einfachste Form bestünde aus einem einzelnen Feature und würde für alle Objekte stehen, die die entsprechende Eigenschaft besitzen, also Träger des jeweiligen Merkmals sind.

Features und Feature-Terme werden in Tabelle I.1 charakterisiert.

Begriff	Bedeutung
Feature	Paar (Feature-Name:Feature-Wert) funktionaler Natur
Feature Term	Konstrukt aus Features mit Mengenoperatoren

Tabelle I.1 *Grundbegriffe der Feature-Logik*

Für die Konstruktion von Feature-Termen aus Features können verschiedene *Mengenoperatoren* verwendet werden. Die entsprechenden syntaktischen Konstrukte für Feature-Terme und deren Semantik sind in Tabelle I.2 aufgelistet.

[1] s. für Anwendungen etwa a. [Smolka, 1992] und [Zeller & Snelting, 1996, Zeller, 1996]

Notation	Name	Interpretation
a	Literal	Menge die a beinhaltet
V	Variable	
$\top$ oder []	Decke	Universalmenge; Konsistenz
$\bot$ oder $\{\}$	Boden	Leere Menge; Inkonsistenz
$f : S$	Auswahl	Der Wert von f ist S
$f \uparrow$	Existenz	f ist definiert
$f \downarrow g$	Übereinstimmung	f und g haben den gleichen Wert
$f \uparrow g$	Differenz	f und g haben verschiedene Werte
$S \sqcap T$ oder $[S, T]$	Schnitt	S und T sind wahr
$S \sqcup T$ oder $\{S, T\}$	Vereinigung	S oder T sind wahr
$\sim S$	Komplement	S ist unwahr
$S \rightarrow T$	Implikation	wenn S wahr ist, dann auch T

Tabelle I.2
Syntax und Semantik von Feature-Termen

Durch Anwendung der Operatoren können eine Vielzahl semantischer Aussagen modelliert werden. Bestimmte Objektmengen werden etwa typischerweise durch Aufzählung ihrer Eigenschaften bzw. Konjunktion der Features ($\sqcap$ oder $[\ldots]$) *selektiert*. Negationen ($\sim$) und Disjunktionen ($\sqcup$ oder $\{\ldots\}$) erlauben die Auswahl von *Alternativen*. Weitere Bedeutungen ergeben sich aus der jeweiligen Anwendungsdomäne.

Finden die vorgegebenen Eigenschaften eines Feature-Terms innerhalb der Objektmenge eine Entsprechung, wird dieser als *konsistent*, andernfalls als *inkonsistent* bezeichnet.

Die durch Feature-Terme beschriebenen Objektmengen erben die Eigenschaften ihrer Bestandteile, die jedoch *angeglichen* bzw. *unifiziert* werden müssen, da Features grundsätzlich funktionaler Natur sind, d.h. dass sie bezüglich eines Objekts genau einen Wert besitzen.

I.2 Rollen

Der funktionale Charakter von Features im mathematischen Sinne erlaubt einerseits deren Einsatz zur Selektion von Objektmengen und ist somit Grundlage des Konsistenzbegriffs. Andererseits sind aber nicht alle interessierenden Eigenschaften funktionaler Natur. Das Konzept der Feature-Logik wurde daher von Smolka [Smolka, 1992] durch Elemente *deskriptiver Logik* generalisiert. Deskriptive Logik ergänzt die grundlegende Feature-Logik dabei um mengenbewertete Eigenschaften, die als *Rollen* (*Roles*) bezeichnet werden. Jede Rolle eines Objektes enthält eine Menge von *Konzeptbeschreibungen* (*Concept Descripti-*

ons), die die Syntax und Semantik der grundlegenden Feature-Logik (Features, Feature-Terme) erben. Bedingungen über diesen Rollen werden mittels Quantoren ausgedrückt (vgl. Abbildung I.3).

Notation	Name	Interpretation
$\forall r(S)$	Universale Quantifikation	Alle Werte von r sind in S
$\exists r(S)$	Existentielle Quantifikation	Mindestens ein Wert von r ist in S

Tabelle I.3
Rollen-Quantoren in Konzeptbeschreibungen

Der *Existenzquantor* $\exists r(S)$ hat die Bedeutung »S enthält mindestens einen Wert von r«. Der Ausdruck $\exists Nachricht(\uparrow Start)$ würde somit alle Objekte umschreiben, die eine Nachricht »$\uparrow Start$« besitzen. Der Ausdruck

$$[\exists Nachricht(\uparrow Start), \exists Nachricht(\uparrow Stop)]$$

würde sich dann auf alle Objekte beziehen, deren Nachrichtenmenge »$\uparrow Start$« *und* »$\uparrow Stop$« beinhaltet. Der *Allquantor* $\forall r(S)$ mit der Bedeutung »S enthält alle Werte von r« kann hingegen als *Filter* verwendet werden: Der folgende Term

$$\forall Nachricht(\uparrow Start \sqcup \uparrow Stop) = \sim \exists Nachricht(\sim\uparrow Start \sqcap \sim\uparrow Stop)$$

selektiert in diesem Sinne grundsätzlich alle Objekte ausser denen, die weder »$\uparrow Start$« noch »$\uparrow Stop$« enthalten.

Funktionale Features werden in dieser Sicht als Spezialfall von Rollen mit jeweils genau einem Wert betrachtet und können so konsistent in das Schema integriert werden.

I.3 Beziehungen

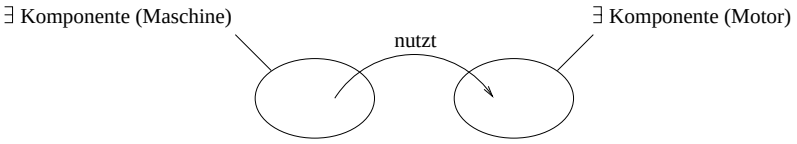
Beziehungen zwischen Objektmengen lassen sich durch Merkmale von Objektmengen in Form von Features bzw. Rollen im Sinne der obigen Ausführungen darstellen. Eine Beziehung zwischen zwei Objektmengen A und B wird im wesentlichen dadurch modelliert, dass eine Rolle von A die Objektmenge B als Wert enthält. Es lassen sich so beliebige n-zu-m-Beziehungen beschreiben. Der folgende Ausdruck zeigt ein Beispiel:

$$A = [\exists Komponente(Maschine), nutzt : \exists Komponente(Motor)]$$

Der Term A repräsentiert die eins-zu-eins Beziehung »$nutzt$« eines Verbundes zwischen Toplevelkomponenten $Maschine$ und Teilkomponenten $Motor$.

Die entsprechenden Komponentenmengen können anschaulich als *Venn-Diagramme* dargestellt werden. Beziehungen sind dann gerichtete Graphen zwischen den Mengen (s. Abbildung I.1).

Abbildung I.1
Beziehungsdiagramm



Beziehungen dieser Art sind existentiell mit den beteiligten Objektmengen verbunden, d.h. ist eine der Rollen inkonsistent, so gilt das auch für die Beziehung. Dies hat zur Folge, dass zum einen nur konsistente Beziehungen – also solche mit nicht-leeren Ausgangs- und Endmengen – selektiert werden können, und zum anderen die Auswahl von Objekten auch immer deren Beziehungen mit einbezieht, wie es insbesondere für eine Komponentensicht auch wünschenswert ist.

Im Zusammenhang mit solchen Beziehungen ist der Begriff der »*vollständigen Konfigurationen*« von wesentlicher Bedeutung. Eine Konfiguration (= Objektmenge, die sich aus einem Ausdruck der Feature-Logik ergibt) ist bezüglich einer Beziehung genau dann *vollständig* bzw. *konsistent*, wenn alle über diese Beziehung referenzierten Objekte auch in der Konfiguration enthalten sind. Für Konfigurationen, die bezüglich einer bestimmten Beziehung *inkonsistent* sind, existieren Methoden *automatischer Erweiterung* zur nachträglichen Vervollständigung [Carpenter, 1992].

I.4 Feature-Strukturen

Terme der Feature Logik lassen sich als Merkmalsstrukturen (engl. *Feature Structures*) repräsentieren. Aus der entsprechenden Darstellung lässt sich besser eine Implementation ableiten, als beispielsweise aus Mengendiagrammen. Im Folgenden ist exemplarisch die Semantikrepräsentation des *Überweisungsauftrags* aus Abschnitt D.2.2 in ihrer Notation als Feature Structure in Form einer Attribut-Wert-Matrix gezeigt, wie sie sich häufig in der Literatur finden (vgl. etwa [Carpenter, 1992])[2] und auch im

[2] S. etwa a. [Wienberg, 2001] für eine breitere Diskussion und der Ausdrucksstärke möglicher Visualisierungen.

Rahmen der Visualierung für die Typmanager-Implementation verwendet wird.

$$\text{TRANSFER}\begin{bmatrix} \text{EMPLOYER} & \begin{bmatrix} \text{NAME} & \text{String} \\ \text{ACCOUNT NUMBER} & \text{Long} \\ \text{BANK CODE NUMBER} & \text{Long} \end{bmatrix} \\ \text{RECEIVER} & \begin{bmatrix} \text{NAME} & \text{String} \\ \text{ACCOUNT NUMBER} & \text{Long} \\ \text{BANK CODE NUMBER} & \text{Long} \end{bmatrix} \\ \text{PURPOSE} & \text{String} \\ \text{AMOUNT} & \text{Double} \end{bmatrix}$$

Abbildung I.2
Feature Structure des Überweisungsauftrags

Abbildungsverzeichnis

1.1 Rollenverteilung während eines Kommunikationsvorganges in der Komponenteninteraktion 6
1.2 Konnektoren ohne oder mit implementierter Adaptionslogik ermöglichen die Überwindung von Inkompatibilitäten 7
1.3 Visuelle Programmierung mit Java Studio-Beans 13
1.4 Anwendungserstellung durch Komposition mittels Komponentenkopplung festgelegter Schnittstellen 19
1.5 Softwarekomponenten 21
1.6 Der Baustoffzyklus 26

2.1 Grundlegende Komponentenstrukturen 58
2.2 Applikationsübergreifende Komponentennutzung 59
2.3 Zyklen 60
2.4 Mögliche Strukturen bei multiplen Interfaces 63
2.5 Bidirektionaler Nachrichtenaustausch 64
2.6 Kanalabstraktion 65
2.7 Schnittstellenspezifikation 76
2.8 Synchronisierte Einzel- und Multischnittstellenspezifikation 78
2.9 Zentralspezifikation 79
2.10 Externe Importspezifikation 80
2.11 Atomare Komponente 91
2.12 Der Komponentenkern 92
2.13 Beispiel einer Schnittstellenstruktur 93
2.14 Komponentenkomposition aus Modellsicht 95
2.15 Komponenteninteraktion als Nachrichtensequenz 96
2.16 Globale Interaktion als Wechselwirkung lokaler Verhaltensaspekte 97
2.17 Formen temporaler Logik 98
2.18 Lineare Zeitlinie mit diskreten Zeitpunkten 99
2.19 Exemplarische Komponentenspezifikation »Maschine« ... 101
2.20 Mögliche korrekte Nachrichtensequenz der Maschine 102
2.21 Verifikation einer Komponentenstruktur 104
2.22 Modifikation des Spezifikationsmodells 106
2.23 Exportbasierte Spezifikation 109
2.24 Anwendung der Feature Logik zur Komponentenstrukturbeschreibung 112
2.25 Multi-Level Modell mit beliebigen Strukturen 113

2.26 Rollenverteilung der Logiken 116

3.1 Nachrichtenaustausch mittels Interzeptoren 135
3.2 Das Auftreten einer (potenziell endlosen) Schleife im Kommunikationsablauf 141
3.3 Ablaufdiagramm des Algorithmus zur Kompatibilitätsprüfung 146
3.4 Ablaufdiagramm des Algorithmus zur Suche nach Nachfolgezustandspaaren 147
3.5 Nachrichtenaustausch mittels Interzeptoren über eine Adapterkomponente 148
3.6 Symmetrische und asymmetrische Kommunikation zweier Komponenten mit einer Adapterkomponente 157
3.7 Prinzip des Policymanagements 170

4.1 Merkmalsdiagramm des *Kreis*-Constraint 187
4.2 drei-stufiges Typmodell 192
4.3 Typarten 199

5.1 Screenshot des Merkmalseditors 243
5.2 Spektrum der Entwicklungsansätze 250
5.3 Aufbau eines Generators in Analogie zu einem Compiler . 264
5.4 Subjektive Schnittstellen bei GenVoca 268
5.5 Schematischer Aufbau eines Generators zur Matrix-Erstellung 272
5.6 Der Generator erhält verschiedene Eingaben und kann Applikationen einer ganzen Produktlinie liefern. 277
5.7 Phasen der Generierung 280
5.8 Quellen des Wissens zur Generierung 281
5.9 Der Ablauf eines Generierungsvorganges 295
5.10 Der Aufbau des grafischen Wizards 296
5.11 Baumstruktur eines konkreten Taschenrechners 304
5.12 a) Alle Typen wurden ausgewählt b) Wenn Typ C optional ist, aber Typ D trotzdem benötigt wird 306
5.13 Grundkonzeption einer generativen Systemerstellung ... 309
5.14 Prinzipsicht der generativen Entwicklungsumgebung 313

6.1 Nachrichtenaustausch mittels `Extension`-Objekten 320
6.2 Synchronisierter Kommunikationsablauf mit Vermittler-Komponente 323
6.3 Einbindung des Adapters in den Kommunikationsablauf . 329
6.4 Die Bestandteile des CCM-Meta-Informationsmodells 350
6.5 Baumdurchlauf bei Tiefensuche 360
6.6 Ablauf des Abbildungsalgorithmus 361
6.7 Typregistrierung im *Type Manager Explorer* 362
6.8 Typoperationen des *Type Manager Explorer* 372
6.9 Ablauf des Prüfalgorithmus 375
6.10 (beidseitig-) vollständig kompatibler Adapter 376

6.11 Suchfunktionen des *Type Manager Explorer* 378
6.12 Ansicht der Typbeschreibung im *Type Manager Explorer* . 380
6.13 Segmentierter Aufbau des Typmanagers 383
6.14 Komponenten des Generators 385
6.15 Ablauf der Merkmalsauswahl 396
6.16 Beispiel wie der XML-DocBuilder arbeitet 401
6.17 Hauptansicht des Deskriptor-Editors im EJB-Modus 415
6.18 Bearbeitung eines Assembly-Deskriptors im EJB-Modus . 417
6.19 Hauptansicht des Protokoll-Editors und seiner Menüs ... 418
6.20 Vorbereitung der Adapterinformationen im Protokoll-Editor 419
6.21 Ansicht des Merkmalseditors 420
6.22 Weitere Darstellungen des Merkmalseditors 422
6.23 Beziehung zum Type-Explorer über Typnamen 423

D.1 Komponententyp *Bank* 453
D.2 Komponententyp *Savings Bank* 454
D.3 Komponententyp *Customer* 456
D.4 Merkmalsdiagramm eines *Überweisungsauftrags* 458
D.5 Merkmalsdiagramm eines *Kundenzugangs* 459
D.6 Merkmalsdiagramm eines *Kundenabgangs* 459
D.7 Assoziationen zwischen *Savings Bank* und *Bank* 462
D.8 Assoziationen zwischen *Customer* und *Bank* 464
D.9 typkonvertierbare (polymorphe) *Bank*-Komponente 467
D.10 Komponententyp *Postscript Print Service* 468
D.11 Komponententyp *Print Service* 469
D.12 Komponententyp *Address Service* 471
D.13 Merkmalsdiagramm eines *Druckauftrags* 473
D.14 Merkmalsdiagramm eines *Druckberichts* 473
D.15 Assoziationen zwischen *Postscript Print Service* und *Print Service* 475
D.16 Assoziationen zwischen *Address Service* und *Postscript Print Service* 477

E.1 Klassendiagramm des Typmodells 479

H.1 Bedeutung von PLTL Operatoren über der Zeitlinie 486

I.1 Beziehungsdiagramm 494
I.2 Feature Structure des *Überweisungsauftrags* 495

Tabellenverzeichnis

4.1 Affinitätsebenen 199
4.2 Assoziationen auf semantischer Ebene 202
4.3 Assoziationen auf Nachrichtenebene 203
4.4 Assoziationen auf syntaktischer Ebene 204
4.5 Vergleichsaussagen für den Komponententyp 206
4.6 Vergleichsaussagen für den syntaktischen Business-Event-Typ 208
4.7 Vergleichsaussagen für den syntaktischen Nachrichtentyp 211

5.1 Komponententypen des Taschenrechners 302
5.2 Nachrichten der Interaktionsmuster der Domäne Taschenrechner 303

6.1 Ablauf der Erstellung eines XML-Baumes 401

G.1 Assoziationen auf semantischer Ebene 483
G.2 Assoziationen auf Nachrichtenebene 484
G.3 Assoziationen auf syntaktischer Ebene 484

H.1 Temporale Operatoren 485

I.1 Grundbegriffe der Feature-Logik 491
I.2 Syntax und Semantik von Feature-Termen 492
I.3 Rollen-Quantoren in Konzeptbeschreibungen 493

Literaturverzeichnis

[**Abadi & Cardelli, 1996**] Abadi, M. und Cardelli, L. *A Theory of Objects*. Springer-Verlag, 1996.

[**Aksit & Tekinerdogan, 1998**] Aksit, M. und Tekinerdogan, B. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters - AOP'98 workshop position paper. Technischer Bericht CS-93-41, TRESE project, University of Twente, Centre for Telematics and Information Technology, P.O. Box 217, 7500 AE, Enschede, The Netherlands, 1998.

[**Allen & Garlan, 1994**] Allen, R. und Garlan, D. Formalizing Architectural Connection. In: *Sixteenth International Conference on Software Engineering*, S. 71–80, Mai 1994.

[**Allen, 1997**] Allen, R. J. *A Formal Approach to Software Architecture*. Dissertation, Carnegie Mellon University, 1997. Report Number CMU-CS-97-144.

[**Arapis, 1991**] Arapis, C. Temporal specifications of object interactions. In: *Proceedings Third International Workshop on Foundations of Models and Languages for Data and Objects*, S. 15–35, Aigen, Austria, Sep 1991.

[**Arapis, 1992**] Arapis, C. *Dynamic Evolution of Object Behaviour and Object Cooperation*. Dissertation, Centre Universitaire d´Informatique, University of Geneva, 1992.

[**Arapis, 1995**] Arapis, C. *A Temporal Perspective of Composite Objects*, Kapitel 5, S. 123–152. In: [Nierstrasz & Tsichritzis, 1995], 1995.

[**AT&T et al., 1996**] AT&T, DSTC, DEC, HP, ICL, Nortel und Novell. Trading Object Service. RFP5 Submission orbos/96–05–06, Version 1.0, Mai 1996.

[**Awizen, 2000**] Awizen, M. Generischer Datenbankzugriff durch XML-Schemata. Diplomarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 2000.

[**Barbacci et al., 1998**] Barbacci, M. R., Carriere, S. J., Feiler, P. H., Kazman, R., Klein, M. H., Lipson, H. F., Longstaff, T. A. und Weinstock, C. B. Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis. Technical Report CMU/SEI-97-TR-029, Carnegie Mellon University SEI, 1998.

[**Bass et al., 1999**] Bass, L., Campbell, G., Clements, P., Northrop, L. und Smith, D. Third Product Line Practice Workshop Report.

Technical Report CMU/SEI-99-TR-003, Carnegie Mellon University SEI, 1999.

[**Bass et al., 1998**] Bass, L., Chaste, G., Clements, P., Northrop, L., Smith, D. und Withey, J. Second Product Line Practice Workshop Report. Technical Report CMU/SEI-98-TR-015, Carnegie Mellon University SEI, 1998.

[**Bass et al., 1997**] Bass, L., Clements, P., Northrop, S. C. L., und Withey, J. Product Line Practice Workshop Report. Technical Report CMU/SEI-97-TR-003, Carnegie Mellon University SEI, 1997.

[**Bass & Kazman, 1999**] Bass, L. und Kazman, R. Architecture-Based Development. Technical Report CMU/SEI-99-TR-007, Carnegie Mellon University SEI, 1999.

[**Batory, 1997**] Batory, D. Intelligent Components and Software Generators. Technical report, Department of Computer Sciences, University of Texas at Austin, 1997. Invited presentation to the Software Quality Institute Symposium on Software Reliability, Austin, Texas.

[**Batory, 1998**] Batory, D. Product-Line Architectures. Eingeladener vortrag, Department of Computer Sciences, University of Texas at Austin, October 1998. Konferenz: Smalltalk und Java in Industrie and Ausbildung, Erfurt, Germany.

[**Batory et al., 1999a**] Batory, D., Chen, G., Robertson, E., und Wang, T. Design Wizards and Visual Programming Environments for GenVoca Generators. 1999.

[**Batory et al., 1994**] Batory, D., Dasari, S., Geraci, B., Singhal, V., Sirkin, M. und Thomas, J. Achieving Reuse With Software System Generators. 1994.

[**Batory & Geraci,**] Batory, D. und Geraci, B. J. Composition Validation and Subjectivity in GenVoca Generators.

[**Batory & Geraci, 1995**] Batory, D. und Geraci, B. J. Validating Component Compositions in Software System Generators. Technical Report 95-03, University of Texas, CSD, 1995.

[**Batory & Geraci, 1997**] Batory, D. und Geraci, B. J. Composition Validation and Subjectivity in Gen Voca Generators. Technical report, Department of Computer Sciences, University of Texas at Austin, 1997.

[**Batory et al., 1997**] Batory, D., Miranker, D. und Brant, D. Jakarta: A Tool Suite for Constructing Software Generators. Technical report, Department of Computer Sciences, University of Texas at Austin, 1997.

[**Batory & Smaragdakis, 2000**] Batory, D. und Smaragdakis, Y. Object-Oriented Frameworks and Product-Lines. 2000.

[**Batory et al., 1999b**] Batory, D., Smaragdakis, Y. und Coglianese, L. Architectural Styles as Adaptors. 1999.

[**Benthien, 1999**] Benthien, T. Versionsbasiertes Komponentenmanagement auf der Basis des MS Component Object Model. Diplomar-

beit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 1999.

[**Bernhardt & Djokić, 2001**] Bernhardt, A. und Djokić, D. Deskriptorbasierte Softwarekonstruktion. Diplomarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 2001.

[**Blando, 2000**] Blando, L. http://www.ccs.neu.edu/home/lblando/ personalities, 2000.

[**Boehm, 1988**] Boehm, B. A spiral model of software development and enhancement. *IEEE Software*, 25(5):61–72, May 1988.

[**Booch, 1987**] Booch, G. *Software Components with ada: Structures, Tools and Subsystems*. Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[**Booch et al., 1998**] Booch, G., Rumbaugh, J. und Jacobson, I. *Unified Modeling Language User Guide*. Addison–Wesley, 1998. s.a. `www.rational.com/uml/index.html`.

[**Bott & Ratcliffe, 1992**] Bott, F. und Ratcliffe, M. *Reuse and Design*, S. 35–51. In: [Hall, 1992], 1992.

[**Braun, 1994**] Braun, C. L. NATO standard for the development of reusable software components. Technischer Bericht Teil 1 von 3, NATO, 1994.

[**Broschei, 2000**] Broschei, N. Konzeption und Realisierung eines Konfigurationswerkzeuges für EJB-Komponenten. Diplomarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 2000.

[**Broy, 1996**] Broy, M. Towards a mathematical model of a component and its use. In: *Componentware Users Conference 1996, Munich, Proceedings*. SIGS Publications, to appear, 1996.

[**Broy, 1998**] Broy, M. Compositional refinement of interactive systems modelled by relations. In: *International Symposium Compositionality 1997*. Lecture Notes in Computer Science, Springer, to appear, 1998.

[**Broy et al., 1998**] Broy, M., Deimel, A., Henn, J., Koskimies, K., Plasil, F., Pomberger, G., Pree, W., Stahl, M. und Szyperski, C. What characterizes a (software) component? *Software Concepts & and tools*, 19(1):49–56, 1998.

[**Büchi & Weck, 1997**] Büchi, M. und Weck, W. A plea for grey-box components. Technischer Bericht TUCS Technical Report No 122, Turku Centre for Computer Science (TUCS), 1997.

[**Burkhardt, 1999**] Burkhardt, D. R. *UML - Unified Modelling Language, Objektorientierte Modellierung für die Praxis*. Addison Wesley, 1999.

[**Burkhart, 1997**] Burkhart, H. *Parallele Programmierung*, S. 479–503. In: [Rechenberg & Pomberger, 1997], 1997.

[**Cardelli, 1989**] Cardelli, L. Typeful Programming. Technical Report 45, DEC SRC Research, 1989.

[**Cardelli & Wegner, 1985**] Cardelli, L. und Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[**Carpenter, 1992**] Carpenter, B. *The logic of typed feature structures – with applications to unification grammars, logic programs and constraint resolution*. Cambridge Press, 1992.

[**Cheong & Jarzabek, 1999**] Cheong, Y. C. und Jarzabek, S. Frame-based method for customizing generic software architectures. In: *Symposium on Software Reusability, SSR99*, S. 103–112. Springer-Verlag, 1999.

[**Christiansen & Münke, 1995**] Christiansen, B. und Münke, M. Diensttypmanagement in offenen verteilten Systemen unter besonderer Berücksichtigung von DCE und CORBA. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 1995.

[**Christiansen & Münke, 1996**] Christiansen, B. und Münke, M. Typmanagement–Unterstützung für Trading in heterogenen verteilten Systemumgebungen: Entwurf und Implementation einer Typmanagement–Komponente. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, FBE Verteilte Systeme, 1996.

[**Clements & Northrop, 1996**] Clements, P. und Northrop, L. Software Architecture: An Executive Overview. Technical Report CMU/SEI-96-TR-00, Carnegie Mellon University SEI, 1996.

[**Clements & Northrop, 1997**] Clements, P. und Northrop, L. A Framework for Software Product Line Practice - Version 2.0. Technischer bericht, Carnegie Mellon University, Pittsburgh, 1997.

[**Clements, 1996**] Clements, P. C. Coming Attractions in Software Architecture. Technical Report CMU/SEI-96-TR-008, Carnegie Mellon University SEI, 1996.

[**Clements & Weiderman, 1998**] Clements, P. C. und Weiderman, N. Report on the Second International Workshop on Development and Evolution of Software Architectures for Product Families. Special Report CMU/SEI-98-SR-003, Carnegie Mellon University SEI, 1998.

[**Cohen, 1999**] Cohen, S. Guidelines for Developing a Product Line Concept of Operations. Technical Report CMU/SEI-99-TR-008, Carnegie Mellon University SEI, 1999.

[**Cohen & ans Spencer Peterson ans James Whithey, 1996**] Cohen, S. und ans Spencer Peterson ans James Whithey, R. K. Models for Domains and Architectures: A Prescription for Systematic Software Reuse. Technischer bericht, Carnegie Mellon University SEI, 1996.

[**Commission, 1995**] Commission, I. E. *Basic Reference Model of Open Distributed Processing — Part3: Architecture. International Standard 10746-3*. International Organisation for Standardization, 1995.

[**Coulouris et al., 1994**] Coulouris, G. F., Dollimore, J. und Kindberg, T. *Distributed Systems - Concepts and Design*. Addisson-Wesley, second Auflage, 1994.

[**Czarnecki, 1996**] Czarnecki, K. Separation of Concerns. *Objektspektrum*, 1996.

[**Czarnecki, 1999**] Czarnecki, K. Generative und komponentenbasierte Softwareentwicklung. Technischer bericht, DaimlerChrysler Forschungszentrum in Ulm und TU Ilmenau, 1999.

[**Czarnecki & Eisenecker, 2000**] Czarnecki, K. und Eisenecker, U. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000.

[**Dingel et al., 1997**] Dingel, J., Garlan, D., Jha, S., und Notkin, D. Towards a formal treatment of implicit invocation. Preprint, Carnegie Mellon University and University of Washington, 1997.

[**Dodani & Tsai, 1992**] Dodani, M. und Tsai, C.-S. ACTS: A Type System for Object–Oriented Programming Based on Abstract and Concrete Classes. In: Lehrman Madsen, O., Hrsg., *European Conference on Object–Oriented Programming, Utrecht, The Netherlands*, Band 615 der *LNCS*, S. 309–324. Springer–Verlag, Juni/Juli 1992.

[**D'Souza & Wills, 1998a**] D'Souza, D. und Wills, A. C. *Catalysis: Objects, Frameworks and Components with UML*. Addison-Wesley, 1998.

[**D'Souza & Wills, 1998b**] D'Souza, D. und Wills, A. C. *Objects, Components, and Frameworks with UML: The Catalysis approach*. Addison Wesley, 1998. http://www.trireme.com/catalysis/book.

[**Dunn & Knight, 1993**] Dunn, M. und Knight, J. Certification of reusable software parts. Technischer Bericht CS-93-41, University of Virginia, 1993.

[**Düwel & Hesse, 2000**] Düwel, S. und Hesse, W. Bridging the gap between Use Case Analysis and Class Structure Design by Formal Concept Analysis. In: Ebert, J. und Frank, U., Hrsg., *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik*, Koblenzer Schriften zur Informatik, S. 27–40. Fölbach Verlag, 2000.

[**Eisenecker & Czarnecki, 1999**] Eisenecker, U. W. und Czarnecki, K. Generatorbau in C++. *Objektspektrum*, 6, 1999.

[**Eisenecker et al., 1997**] Eisenecker, U. W., Czarnecki, K. und Steyaert, P. Beyond Objects: Generative Programming. Technischer bericht, Position Paper ECOOP '97 Workshop on Aspect-Oriented Programming, 1997.

[**Fisher & Nielsen, 1994**] Fisher, K. und Nielsen, M. Notes on typed object-oriented programming. In: *Lecture Notes in Computer Science: Proceedings TACS'94*, S. 844–885. Springer-Verlag, 1994.

[**Fowler et al., 1999**] Fowler, M., Beck, K., Brant, J., und Opdyke, W. *Refactoring*. Addison-Wesley, 1999. ISBN 0201485672.

[**Fowler et al., 2000**] Fowler, M., Beck, K., Brant, J., Opdyke, W. und Roberts, D. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 2000.

[**Franch, 1997**] Franch, X. The convenience for a notation to express non-functional characteristics of software components. S. 101–110, Sep 1997.

[**Friedrich et al., 1995**] Friedrich, J., Herrmann, T., Peschek, M., und (Hrsg.), A. R. *Informatik und Gesellschaft*. Spektrum Akademischer Verlag, Heidelberg; Berlin; Oxford, 1995.

[**Gabbay, 1992**] Gabbay, D. Temporal logic: Mathematical foundations. Technischer Bericht MPI-I-92-213, Max-Planck-Institut für Informatik, Im Stadtwald, W 6600 Saarbrücken, Germany, Mar 1992.

[**Gabbay et al., 1992**] Gabbay, D., Hodkinson, I. und Reynolds, M. Temporal logic: Mathematical foundations, part 2. Technischer Bericht MPI-I-92-242, Max-Planck-Institut für Informatik, Im Stadtwald, W 6600 Saarbrücken, Germany, Sep 1992.

[**Gamma et al., 1994**] Gamma, E., Helm, R., Johnson, R., und Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

[**Garlan, 1995**] Garlan, D. Research Directions in Software Architecture. *ACM Computing Surveys*, 27(2):257–261, 1995.

[**Garlan et al., 1994**] Garlan, D., Allen, R. und Ockerbloom, J., Hrsg. *Exploiting Style in Architectural Design Environments*, 1994.

[**Garlan et al., 1995a**] Garlan, D., Allen, R. und Ockerbloom, J. Architectural Mismatch or Why it's hard to build systems out of existing parts. In: *Proceedings of the 17th International Conference on Software Engineering. Seattle, April 1995*, S. 170–185. ACM Press, 1995.

[**Garlan & Métayer, 1997**] Garlan, D. und Métayer, D. L., Hrsg. *Coordination Languages and Models, Proceedings of the 2nd International Conference, COORDIANTION'97, Berlin*, Nummer 1282 der Lecture Notes in Computer Science. Springer–Verlag, September 1997.

[**Garlan et al., 1995b**] Garlan, D., Monroe, R. und Wile, D. ACME: An Architectural Interconnection Language. Technischer bericht, Carnegie Mellon University, November 1995.

[**Garlan et al., 1997**] Garlan, D., Monroe, R. und Wile, D. Acme: An Architecture Description Interchange Language. Proceedings of CASCON'97, 1997.

[**Gluch & Weinstock, 1998**] Gluch, D. P. und Weinstock, C. B. Model-Based Verification: A Technology for Dependable System Upgrade. Technical Report CMU/SEI-98-TR-009, Software Engineering Institute, Carnegie Mellon University, 1998.

[**Göllnitz, 1997**] Göllnitz, M. Generische Verarbeitung von Richtlinien für offene verteilte Systeme. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1997.

[**Griffel, 1996**] Griffel, F. Spezifikation und Realisierung von Interoperabilitätskonzepten zur Integration heterogener verteilter Systemplattformen. Diplomarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 1996.

[**Griffel, 1998**] Griffel, F. *Componentware – Konzepte und Techniken eines Softwareparadigmas*. dpunkt.verlag, 1998.

[**Griffel, 1999a**] Griffel, F. Application-Server als Anwendungsdrehscheibe. *Computerwoche focus*, (5), September 1999.

[**Griffel, 1999b**] Griffel, F. BOCA - Konzepte und Techniken der OMG Business Object Component Architecture. In: *Component Development Europe, Frankfurt (CD'99)*. SIGS, 1999.

[**Griffel, 1999c**] Griffel, F. Distributed Object Computing with Java/CORBA Components. In: *Workshop der ObjectWorld'99, Mai 1999, Berlin*, 1999.

[**Griffel, 1999d**] Griffel, F. Evolutionäre und adaptive Verfahren der Softwarekonstruktion. Seminar der Nordakademie, Elmshorn, Februar 1999.

[**Griffel, 1999e**] Griffel, F. Inside Microsoft (D)COM++. In: *ObjectWorld'99, Berlin, Mai 1999*, 1999.

[**Griffel, 1999f**] Griffel, F. Kommunikationsbasierte Integration heterogener Softwarekomponenten. In: *Component Development Europe, Frankfurt, September 1999*, 1999.

[**Griffel, 1999g**] Griffel, F. Komponententechnologien - Standortbestimmung und Perspektiven. In: *Informatikforum'99, Heidelberg, Mai 1999*, 1999.

[**Griffel, 1999h**] Griffel, F. Metainformationen und -modelle in der Komponentensicht: Konzepte und Techniken. In: *KnowTechForum'99, Potsdam*, 1999.

[**Griffel, 1999i**] Griffel, F. Qualitätsmetriken in der komponentenbasierten Softwareentwicklung. Seminar des PASS Beraterforum, Frankfurt, September 1999.

[**Griffel, 1999j**] Griffel, F. Skalierung des Microsoft Component Object Model. *Computerwoche*, 1999. 21.5.1999.

[**Griffel, 1999k**] Griffel, F. Softwarekomponenten als Hoffnungsträger. *Computerwoche*, 1999. 21.5.1999.

[**Griffel, 2000a**] Griffel, F. Einsatzmöglichkeiten komponentenbasierter Meta-Modelle zur generativen, konsistenzsichernden Softwarekonstruktion. In: Ebert, J. und Frank, U., Hrsg., *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik*, Koblenzer Schriften zur Informatik, S. 71–87. Fölbach Verlag, 2000.

[**Griffel, 2000b**] Griffel, F. Komponentenbasierte Softwareentwicklung – wie steht es um die Bausubstanz? In: Bullinger, H., Hrsg., *Komponentenbasierte Software für Produkte und Dienstleistungen, IAO-Forum der Fraunhofer Gesellschaft, Stuttgart*, 2000.

[**Griffel, 2000c**] Griffel, F. Wie gut ist die Bausubstanz? - Fortgeschrittene Beschreibungstechniken für Softwarekomponenten. *OBJEKT-Spektrum*, (2), 2000.

[**Griffel, 2001**] Griffel, F. *Componentware*. dpunkt. Verlag, 2. Auflage, 2001.

[**Griffel et al., 1996**] Griffel, F., Müller–Jones, K. und Lamersdorf, W. Komponentenbasierte Entwicklung interoperabler Software auf heterogenen Middleware–Plattformen. In: Mayr, H. C., Hrsg., *Beherrschung von Informationssystemen, Tagungsband der Informatik'96*, Nummer 88 der Schriftenreihe der Österreichischen Computer Gesellschaft, S. 327–342. R. Oldenbourg, 1996.

[**Griffel et al., 1999**] Griffel, F., Tu, M. T., Zirpins, C., und Lamersdorf, W. Towards Policy-mediated Component Composition. Preprint coord'99, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 1999.

[**Hall, 1992**] Hall, P. A. V., Hrsg. *Software Reuse and Reverse Engineering in Practice*. Chapman & Hall, 1992.

[**Harrison, 2000**] Harrison, W.; Ossher, H. http://www.research.ibm.com/sop, 2000.

[**Harrison & Ossher, 1993**] Harrison, W. und Ossher, H. Subject–Oriented Programming — A Critique of Pure Objects. In: *Proceedings of the 1993 Conference on Object–Oriented Programming Systems, Languages, and Applications*, September 1993.

[**Herzum & Sims, 2000**] Herzum, P. und Sims, O. *Business Component Factory*. OMG Press. John Wiley & Sons, 2000.

[**Hesse, 2000**] Hesse, W. Software-Projektmanagement braucht klare Strukturen – Kritische Anmerkungen zum »Rational Unified Process«. In: Ebert, J. und Frank, U., Hrsg., *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik*, Koblenzer Schriften zur Informatik, S. 143–150. Fölbach Verlag, 2000.

[**Hühnken, 2001**] Hühnken, L. Temporallogische Verhaltensspezifikationen für das JavaBean-Modell. Studienarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 2001.

[**Häming, 1999**] Häming, A. Eine CORBA-Infrastruktur zur dynamischen Konfiguration komponentenbasierter Anwendungs-Server. Studienarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 1999.

[**Häming, 2000**] Häming, A. Konzeption und Realisierung einer Typmanagementkomponente zur Unterstützung generativer Softwarekonstruktionn. Diplomarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 2000.

[**Hoare, 1978**] Hoare, C. A. R. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug 1978.

[**Holibough & Perry, 1988**] Holibough, R. und Perry, J. Phase I testbed description: Requirements and selection guidelines. Technischer Bericht CMU/SEI-88-TR-13, Software Engineering Institute, Carnegie Mellon University, Sep 1988.

[**Hondt et al., 1997**] Hondt, K. D., Lucas, C. und Steyaert, P. Reuse contracts as component interface descriptions. S. 43–49. Turku Centre for Computer Science General Publication No 5, Sep 1997.

[**Hooper & Chester, 1991**] Hooper, J. W. und Chester, R. O. *Software Reuse: Guidelines and Methods*. Plenum Press, New York, 1991.

[**Hopcroft & Ullman, 1993**] Hopcroft, J. und Ullman, J. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Adison-Wesley, 1993.

[**Hopcroft & Ullman, 1988**] Hopcroft, J. E. und Ullman, J. D. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1988.

[**Horn et al., 1998**] Horn, E., Kupries, M. und Reinke, T. Architekturtypen für Agentenanwendungssysteme. Preprint 001/1998, Universität Potsdam, Institut für Informatik, März 1998.

[**Huber & Seidl, 1999**] Huber, S. und Seidl, H. Architekturzentriertes Entwickeln mit der UML. *Objektspektrum*, 4/99, 1999.

[**Huber & Elting, 1999**] Huber, W. und Elting, A. Catalysis, ein Vorgehensmodell für die komponentenbasierte und objektorientierte Softwareentwicklung. *Objektspektrum*, 6/99, 1999.

[**Jackson & Wing, 1996**] Jackson, D. und Wing, J. Lightweight Formal Methods. *IEEE Computer*, 29(4):21–22, 1996.

[**Jackson, 1995**] Jackson, M. *Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices*. ACM Press, Addison-Wesley, 1995.

[**Jacobsen et al., 1994**] Jacobsen, L., Griss, M. und Johnson, P. *Software Reuse — Architecture, Process and Organisation for Business Success*. ACM Press/Addison Wesley, 1994.

[**Jacobson et al., 1999**] Jacobson, I., Booch, G. und Rumbaugh, J. *Unified Software Development Process*. Addison-Wesley Object Technology Series, 1999. s.a. http://www.rational.com/products/rup/prodinfo/reading/index.jtmpl.

[**Java Beans, 1997**] Java Beans Spezification. Specification 1.01, Sun Microsystems, 1997.

[**Jeffries, 1999**] Jeffries, R. E. eXtreme Testing. *STQE Magazin*, 2, 1999.

[**Josuttis, 1999**] Josuttis, N. eXtreme Programming: ein Gespräch mit Kent Beck. *OBJEKTspektrum*, 4, 1999.

[**Kain, 1996**] Kain, J. B. Components: The basics: Enabling an application or system to be the sum of its parts. *Object Magazine*, 6(2):64–69, Apr 1996.

[**Kazman & Carriére, 1997**] Kazman, R. und Carriére, S. J. Playing Detective: Reconstructing Software Architecture from Available Evidence. Technical Report CMU/SEI-97-TR-010, Carnegie Mellon University SEI, 1997.

[**Kiczales et al., 1997**] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. und Irwin, J. Aspect Oriented

Programming. Technischer bericht, Xerox Palo Alto Research Center, 1997.

[**Kirtland, 1997**] Kirtland, M. The COM+ Programming Model Makes it Easy to Write Components in Any Language. *Microsoft Systems Journal*, Dezember 1997.

[**Klein & Kazman, 1999**] Klein, M. und Kazman, R. Attribute-Based Architectural Styles. Technical Report CMU/SEI-99-TR-022, Carnegie Mellon University SEI, 1999.

[**Klein et al., 1998**] Klein, R. K. M., Barbacci, M., Longstaff, T., Lipson, H. und Carriere, J. The Architecture Tradeoff Analysis Method. Technical Report CMU/SEI-98-TR-008, Carnegie Mellon University SEI, 1998.

[**Konstantas, 1995**] Konstantas, D. *Interoperability of Object-Oriented Applications*, Kapitel 3, S. 69–95. In: [Nierstrasz & Tsichritzis, 1995], 1995.

[**Kröhling, 2000**] Kröhling, T. Generative Komponentenkomposition. Diplomarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 2000.

[**Kruchten, 1995**] Kruchten, P. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[**Krut, Jr., 1993**] Krut, Jr., R. W. Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology. Technical Report CMU/SEI-93-TR-11, Carnegie Mellon University SEI, 1993.

[**Lapok, 2000**] Lapok, M. Nachrichtenbasierte Kopplung von Softwarekomponenten. Diplomarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 2000.

[**Leroy, 1994**] Leroy, X. Manifest types, modules, and separate compilation. In: *Proceedings POPL'94*, S. 109–122. ACM Press, 1994.

[**Levi & Rappaport Hovav, 1995**] Levi, B. und Rappaport Hovav, M. *Unaccusativity: At the syntax-lexical semantics interface*. Nummer 26 der Linguistic Inquiry. MIT, 1995.

[**Lieberherr, 1997**] Lieberherr, K. Demeter and Aspect-Oriented Programming: Why are programs hard to evolve? . Technical report, 1997.

[**Lieberherr, 1996**] Lieberherr, K. J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[**Lions, 1996**] Lions, J. Ariane 5 — flight 501 failure — report by the inquiry board. Technischer bericht, European Space Agency (ESA), Paris, France, Jul 1996.

[**Liskov & Wing, 1993**] Liskov, B. und Wing, J. M. A New Definition of the Subtype Relation. In: Nierstrasz, O. M., Hrsg., *Proceedings of the 7th European Conference on Object–Oriented Programming, ECOOP '93, Kaiserslautern*, Nummer 707 der LNCS, S. 117–141. Springer–Verlag, Juli 1993.

[**Luckham et al., 1995**] Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D. und Mann, W. Specification and Analysis of System Archtitecture Using Rapide. In: *IEEE Transactions on Software Engineering*, S. 336–355, April 1995.

[**Magee et al., 1995**] Magee, J., Dulay, N., Eisenbach, S., und Kramer, J. Specifying Distributed Software Architectures. In: *Proceedings of the Fifth European Software Engineering Conference, ESEC'95*, 1995.

[**Manna & Wolper, 1984**] Manna, Z. und Wolper, P. Synthesis of communicating process. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, Jun 1984.

[**Martens, 1999**] Martens, C. Dynamische Aktualisierungsmechanismen komponentenbasierter Software. Diplomarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 1999.

[**Matena & Hapner, 1997**] Matena, V. und Hapner, M. *Enterprise JavaBeans 1.0 architecture*. Sun Microsystems, 1997.

[**McGregor et al., 1996**] McGregor, J. D., Doble, J. und Keddy, A. A pattern for reuse: Let architectural reuse guide component reuse. *Object Magazine*, 6(2):38–47, Apr 1996.

[**McIlroy, 1968**] McIlroy, M. D. Mass Produced Software Components. In: Naur, P. und Randell, B., Hrsg., *Software Engineering: Report on a Conference by the NATO Science Committee*, S. 138–155, Brüssel, 1968. NATO Scientific Affairs Division.

[**Medvidovic et al., 1997**] Medvidovic, N., Oreizy, P. und N., T. R. Reuse of Off-the-Shelf Components in C2-Style Architectures. In: *Symposium on Software Reusability (SSR97)*, S. 190–198, Mai 1997.

[**Mens et al., 1997**] Mens, K., Steyaert, P. und Lucas, C. *Reuse Contracts: Managing Evolution in Adaptable Systems*, Kapitel I: Proceedings of the ECOOP Workshop on Adaptability in Object-Oriented Software Development, S. 37–42. In: [Mühlhäuser, 1997], 1997.

[**Merz, 1997**] Merz, S. Implementierung eines endlichen Automaten, 1997. `http://www2.informatik.unibw-muenchen.de/Education/cbt/StateMachine/Simulator.html`.

[**Mey, 1995**] Mey, V. *Visual Composition of Software Applications*, Kapitel 10, S. 275–303. In: [Nierstrasz & Tsichritzis, 1995], 1995.

[**Meyer, 1988**] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[**Moehl, 1999a**] Moehl, M. Dynamische Adaptionsmechanismen komponentenorientierter Software. Diplomarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 1999.

[**Moehl, 1999b**] Moehl, M. Evaluation dynamischer Mechanismen komponentenbasierter Software. Studienarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 1999.

[**Moriconi et al., 1995**] Moriconi, M., Qian, X. und Riemenschneider, R. A. Correct Architecture Refinement. In: *IEEE Transactions on Software Engineering*, S. 356–372, April 1995.

[**Mühlhäuser, 1997**] Mühlhäuser, M., Hrsg. *Spezial Issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP'96*. dpunkt.Verlag, 1997.

[**Müller–Jones et al., 1995**] Müller–Jones, K., Merz, M. und Lamersdorf, W. The TRADEr: Integrating Trading Into DCE. In: Raymond, K. und Armstrong, L., Hrsg., *Open Distributed Processing: Experiences with Distributed Environments, Proceedings of the 3rd IFIP TC 6/WG 6.1 International Conference on Open Distributed Processing*, S. 476–487. Chapman & Hall, 1995.

[**Müller et al., 1994**] Müller, K., Merz, M. und Lamersdorf, W. Der TRADE–Trader: Ein Basisdienst offener verteilter Systeme. In: Popien, C. und Meyer, B., Hrsg., *Neue Konzepte für die Offene Verteilte Verarbeitung*, S. 35–44. Verlag der Augustinus Buchhandlung, Aachen, September 1994.

[**Murer et al., 1997**] Murer, T., Scherer, D. und Würtz, A. *Improving Component Interoperability*, Kapitel III: Proceedings of the ECOOP Workshop on Component-Oriented Programming (WCOP'96), S. 150–158. In: [Mühlhäuser, 1997], 1997.

[**Musser & Stepanov, 1989**] Musser, D. R. und Stepanov, A. A. Generic programming. In: *ISSAC '88 Symbolic and Algebraic Computation Proceedings, Lecture Notes in Computer Science 358*. Addison-Wesley, 1989.

[**Musser & Stepanov, 1994**] Musser, D. R. und Stepanov, A. A. Algorithm-oriented generic libaries. 24(7), 1994.

[**Neubert, 1998**] Neubert, T. Anwendung von generativen Programmiertechniken am Beispiel der Matrixalgebra. Diplomarbeit, Technische Universität Chemnitz, 1998.

[**Nierstrasz, 1995a**] Nierstrasz, O. *Regular Types*, Kapitel 4. In: [Nierstrasz & Tsichritzis, 1995], 1995.

[**Nierstrasz, 1995b**] Nierstrasz, O. Regular Types for Active Objects. In: [Nierstrasz & Tsichritzis, 1995], Kapitel 4, S. 3–28.

[**Nierstrasz & Dami, 1995**] Nierstrasz, O. und Dami, L. *Component-Oriented Software Technology*, Kapitel 1, S. 3–28. In: [Nierstrasz & Tsichritzis, 1995], 1995.

[**Nierstrasz & Lumpe, 1997**] Nierstrasz, O. und Lumpe, M. Komponenten, Komponentenframeworks und Gluing. *HMD — Theorie und Praxis der Wirtschaftsinformatik*, 197:8–23, 9 1997.

[**Nierstrasz & Tsichritzis, 1995**] Nierstrasz, O. und Tsichritzis, D., Hrsg. *Object–Oriented Software Composition*. The Object–Oriented Series. Prentice–Hall International Ltd., 1995.

[**Oberndorf et al., 1997**] Oberndorf, P., Brownsword, L., Morris, E., und Sledge, C. Workshop on COTS-Based Systems. Special Report CMU/SEI-97-SR-019, Carnegie Mellon University SEI, 1997.

[**Oestereich, 1998**] Oestereich, B. *Objektorientierte Softwareentwicklung, Analyse und Design mit der Unified Modelling Language*. Oldenbourg, 1998.

[**Oestereich, 1999**] Oestereich, B. Glossar des Object Engineering Process (OEP). Technical Report, 1999.

[**Okumura, 1986**] Okumura, K. A formal protocol conversion method. In: *ACM SIGCOMM '86 Symposium*, S. 30–37. ACM, August 1986.

[**Olafsson & Bryan, 1997**] Olafsson, A. und Bryan, D. *On the Need for "Required Interfaces" of Components*, Kapitel III: Proceedings of the ECOOP Workshop on Component-Oriented Programming (WCOP'96), S. 159–165. In: [Mühlhäuser, 1997], 1997.

[**OMG, 1996a**] OMG. *Common Facilities RFP-4, Common Business Objects and Business Object Facility*. Object Management Group (OMG), Framingham, MA., USA, Jan 1996.

[**OMG, 1996b**] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group (OMG), Framingham, MA., USA, Jul 1996.

[**OMG, 1998**] OMG. Business Object Component Architecture; Revised Proposal, 1998. bom/98-07-01.

[**OMG, 1999**] OMG. CORBA Component Model. OMG TC Document orbos/99-02-05, Object Management Group, März 1999.

[**OMG, 1999a**] OMG. CORBA Components, 1999. orbos/99-02-05.

[**OMG, 1999b**] OMG. Meta Object Facility (MOF) Specification. `www.omg.org`, 1999. Dokument ad/99-07-01.

[**Papathomas, 1995**] Papathomas, M. *Concurrency in Object-Oriented Programming Languages*, Kapitel 2, S. 31–68. In: [Nierstrasz & Tsichritzis, 1995], 1995.

[**Parnas, 1976**] Parnas, D. L. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, TSE2(1):1–9, 1976.

[**Pazzi, 1997**] Pazzi, L. *An Explizit Modelling Perspective for Compound and Aggregate Entities in the Object Paradigm*, Kapitel III: Proceedings of the ECOOP Workshop on Component-Oriented Programming (WCOP'96), S. 166–171. In: [Mühlhäuser, 1997], 1997.

[**Peterson & Stanley, Jr., 1994**] Peterson, A. S. und Stanley, Jr., J. L. Mapping a Domain Model and Architecture to a Generic Design. Technical Report CMU/SEI-94-TR-8, Carnegie Mellon University SEI, 1994.

[**Pree, 1997**] Pree, W. *Komponentenbasierte Softwareentwicklung mit Frameworks*. dpunkt.verlag, 1997.

[**Prehofer, 1997**] Prehofer, C. Feature-Oriented Programming: A Fresh Look at Objects. In: *Proceedings of ECOOP'97*, S. 419–443, 1997.

[**Rath, 1999**] Rath, H. H. Mozart oder kugel. *iX*, (12), 1999.

[**Rechenberg & Pomberger, 1997**] Rechenberg, P. und Pomberger, G., Hrsg. *Informatik-Handbuch*. Carl Hanser Verlag, München Wien, 1997.

[**Rumbaugh et al., 1991**] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. und Lorensen, W. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[**Sametinger, 1997**] Sametinger, J. *Software Engineering with Reusable Components*. Springer-Verlag Berlin Heidelberg New York, 1997.

[**Schmidt, 1997**] Schmidt, M. Generische, orthogonale un persistente Verarbeitung von Eigenschaften: Eine Implementierung des CORBA Property Service in Java. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 1997.

[**Schnell et al., 1996**] Schnell, K., Zalman, N. und Bhatt, A. Transitioning Domain Analysis: An Industry Experience. Technical Report CMU/SEI-96-TR-009, Carnegie Mellon University SEI, 1996.

[**Seacord et al., 1998**] Seacord, R. C., Hissam, S. A. und Wallnau, K. C. Agora: A Search Engine for Software Components. Technical Report CMU/SEI-98-TR-011, Carnegie Mellon University SEI, 1998.

[**SEI, 1993**] SEI. Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology. Technical Report CMU/SEI-93-TR-11, Software Engineering Institute, Carnegie Mellon University, 1993.

[**Shaw, 1990**] Shaw, M. Toward Higher-Level Abstraction for Software Systems. In: *Data and Knowledge Engineering*, Band 5, S. 119–128. North Holland, 1990.

[**Shaw, 1995**] Shaw, M. Making Choices: A Comparison of Styles for Software Architecture. *IEEE Software*, 12(6):27–41, 1995.

[**Shaw, 1996**] Shaw, M. Truth vs. Knowledge: The Difference Between What a Component Does and What We Know It Does. In: *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996.

[**Shaw et al., 1995**] Shaw, M., DeLine, Klein, R. D. V., Ross, T. L., Young, D. M. und Zelesnik, G. Abstractions for Software Architecture and Tools to Support Them. In: *IEEE Transactions on Software Engineering*, April 1995.

[**Simonyi, 1995**] Simonyi, C. The Death Of Computer Languages, The Birth Of Intentional Programming. Technical Report MSR-TR-95-52, Microsoft Corporation, 1995.

[**Simos, 1997**] Simos, M. Organization Domain Modelling and OO Analysis and Design: Distinctions, Integration, New Directions. Technical Report, Organon Motives, Inc, 1997.

[**Smaragdakis, 1999**] Smaragdakis, I. *Implementing Large-Scale Object-Oriented Component*. Dissertation, University of Texas, 1999.

[**Smaragdakis, 1997**] Smaragdakis, Y. DiSTiL : A Transformation Library for Data Structures. Technical report, Department of Computer Sciences, University of Texas at Austin, October 1997. USENIX Conference on Domain-Specific Languages.

[**Smaragdakis & Batory, 1999**] Smaragdakis, Y. und Batory, D. Implementing Reusable Object-Oriented Components. 1999.

[**Smaragdakis & Batory, 2000**] Smaragdakis, Y. und Batory, D. Application Generators. Technical report, Department of Computer Sciences, University of Texas at Austin, 2000. `ftp://ftp.cs.utexas.edu/pub/predator/generators.ps`.

[**Smolka, 1992**] Smolka, G. Feature-constrained logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[**Solderitsch et al., 1995**] Solderitsch, J., Wickman, G., Kweder, D., und Horton, M. An Architecture and Generator for an Army IEW Domain. Technical report, Software Technology Conference, 1995.

[**Spell, 2000**] Spell, B. *Professional Java Programming*. wrox, 2000.

[**Stal, 1995**] Stal, M. Componentware — die Vision von verteilten Objekten. *OBJEKTspektrum*, (4):77–81, Juli/August 1995.

[**Störig, 1998**] Störig, H. J. *Kleine Weltgeschichte der Philosophie*. Fischer, 1998.

[**Sun, 1997a**] Sun. *Java Beans 1.01 API Specification*. Sun Microsystems, Palo Alto, CA., USA, 1997.

[**Sun, 1997b**] Sun. *Java Beans for Java Studio: Architecture and API - A Technical White Paper*. Sun Microsystems, Palo Alto, CA., USA, 1997.

[**Sun, 1999a**] Sun. Java 2 Plattform, Standard Edition, v1.2.2 API Specification. Technischer bericht, Sun Microsystems, 1999.

[**Sun, 1999b**] Sun. Java Project X API Specification. Technischer bericht, Sun Microsystems, 1999.

[**Sun, 2000**] Sun. Enterprise Java Beans Spezification. Specification 1.1, Sun Microsystems, 2000.

[**Szyperski, 1998**] Szyperski, C. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[**Team, 1997**] Team, R. D. *Guide to the Rapide 1.0 Language Reference Manuals*, 1997.

[**Tetau, 1998**] Tetau, D. Qualitätsmetriken komponentenbasierter Software. Studienarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 1998.

[**Tetau, 1999**] Tetau, D. Regelgestützte Komponentenkomposition. Diplomarbeit, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 1999.

[**Tolksdorf, 1999**] Tolksdorf, R. XML und darauf basierende Standards: Die neuen Auszeichnungssprachen des Web. *Informatik Spektrum*, 22(6), 1999.

[**Tu et al., 1997**] Tu, M. T., Griffel, F., Merz, M., und Lamersdorf, W. Generic Policy Management for Open Service Markets. In: König, H., Geihs, K. und Preuß, T., Hrsg., *Distributed Applications and Interoperable Systems, DAIS'97 Cottbus, Germany*, S. 211–222. IFIP, Chapman & Hall, Oktober 1997.

[**Tu, 2000**] Tu, T. *Konzeption und Entwicklung einer Steuerungseinheit zur Ausführung von automatisierten Verhandlungen*. Dissertation, Universität Hamburg, FB Informatik, FBE Verteilte Systeme, 2000.

[**van Hoff et al., 1997**] van Hoff, A., Partovi, H. und Thai, T. The Open Software Description Format(OSD). Technischer bericht, Marimba Incorporated and Microsoft Corporation, 1997. http://www.w3.org/TR/NOTE-OSD.html.

[**Vanhelsuwé, 1997**] Vanhelsuwé, L. *Mastering JavaBeans*. SYBEX, 1997.

[**Vestal, 1996**] Vestal, S. *MetaH Programmer's Manual Version 1.09*. Honeywell Technology Center, 1996.

[**W3C, 1998**] W3C. Extensible Markup Language (XML) 1.0. W3c recommendation, World Wide Web Consortium, 1998.

[**W3C, 2000a**] W3C. http://www.w3c.org/tr/note-dcd, 2000.

[**W3C, 2000b**] W3C. http://www.w3c.org/tr/wd-xlink, 2000.

[**W3C, 2000c**] W3C. http://www.w3.org/tr/rec-dom-level-1, 2000.

[**Wegner, 1993**] Wegner, P. Towards component-based software technology. Technischer Bericht CS-93-11, Brown University, 1993.

[**Wegner, 1997**] Wegner, P. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.

[**Wienberg, 2001**] Wienberg, F. *Informations- und prozessorientierte Modellierung verteilter Systeme auf der Basis von Feature-Structure-Netzen*. Dissertation, Universität Hamburg, FB Informtik, FBE Verteilte Systeme, 2001.

[**Woods et al., 1999**] Woods, S., Carriere, S. und Kazman, R. A Semantic Foundation for Architectural Reengineering and Interchange. In: *Proceedings of the International Conference on Software Maintenance (ICSM-99). Oxford*, S. 391–398. IEEE Computer Society, 1999.

[**Yellin & Strom, 1995**] Yellin, D. M. und Strom, R. E. Collaboration Specifications and Component Adaptors. Forschungsbericht RC 20054 (88710) 5/4/97, T.J. Watson Research Center, Mai 1995.

[**Zeller, 1996**] Zeller, A. *Configuration Management with Version Sets*. Dissertation, Technical University of Braunschweig, Germany, Nov. 1996.

[**Zeller & Snelting, 1996**] Zeller, A. und Snelting, G. Unified versioning through feature logic. Technischer Bericht 96-01, Technical University of Braunschweig, Germany, Mar 1996.

[**Zirpins, 1999**] Zirpins, C. Grundlagen komponentenorientierter Software: Ein Basismodell für Componentware. Diplomarbeit, Universität Hamburg, FB Informtik, FBE Verteilte Systeme, 1999.