Informations- und prozeßorientierte Modellierung verteilter Systeme auf der Basis von Feature-Structure-Netzen

Dissertation

zur Erlangung des Doktorgrades
am Fachbereich Informatik
der Universität Hamburg

vorgelegt von
Frank Wienberg
wienberg@informatik.uni-hamburg.de

1. November 2001

Annahmevermerk

Genehmigt vom Fachbereich Informatik der Universität Hamburg
auf Antrag der Gutachter

Prof. Dr. Winfried Lamersdorf und
Prof. Dr. Rüdiger Valk.

Zusammenfassung

Durch die wachsenden Ansprüche an verteilte Software-Systeme wie beispielsweise Internet-Anwendungen steigt deren Komplexität in eine Dimension, die mit heutigen Mitteln kaum noch zu bewältigen ist. Daher gibt es derzeit einen hohen Bedarf nach Ansätzen, die eine sinnvolle *Spezifikation verteilter Systeme* erlauben. Eine wichtige Basis von Spezifikationsansätzen ist eine *Modellierungstechnik*, welche Syntax und Semantik von Modellen festlegt und deren Erstellung, Analyse und Validation durch entsprechende *Werkzeuge* unterstützt.

Die vorliegende Arbeit stellt *Feature-Structure-Netze* (FSNets) als eine Modellierungstechnik für verteilte Systeme vor, die auf Feature Structures und Petrinetzen basiert. Feature Structures unterstützen die *informationsorientierte* Modellierung, während durch Petrinetze die *prozeßorientierten* Aspekte des Modells spezifiziert werden. Nach einem Blick auf andere informations- und prozeßorientierte Modellierungstechniken mit einem Schwerpunkt auf der *Unified Modeling Language* (UML) werden Syntax und Semantik der FSNets formal definiert. Es wird eine UML-Notation für Feature Structures vorgestellt, die in FSNets eingesetzt wird. Dadurch stellen FSNets den ersten Petrinetzformalismus dar, der konsequent UML zur Modellierung und Darstellung von Typen und Objekten nutzt. Durch ihre *operationale Semantik* können FSNet-Modelle direkt ausgeführt werden. Der Modellierer wird durch ein graphisches Werkzeug unterstützt, das die Bearbeitung und Ausführung von FSNets erlaubt.

Im theoretischen Teil der Arbeit werden Feature Structures als Formalisierung von abstrakten Objektgraphen oder *Objektsichten* untersucht und *Basis-FSNets* sowie vor allem *elementare FSNets* (EFSNets) formal definiert und zur theoretischen Betrachtung spezieller Phänomene verteilter Systeme herangezogen. Als wesentliche Beiträge werden *gefärbte Prozesse* von EFSNets als eine erweiterte Prozeßsemantik für Netze in Netzen untersucht und es wird die Konsistenz verteilter Kopien von Objekten oder Prozessen und damit die Unterscheidung von *Wert- und Referenzsemantik* betrachtet.

Als praktisches Ergebnis der Arbeit werden Basis-FSNets mit Konzepten der Referenznetze zu höheren FSNets (HFSNets) erweitert, um Anforderungen aus realistischen Problemstellungen zu genügen. Es wird eine graphische, UML-basierte Notation und ein Software-Werkzeug zur Erstellung und Ausführung von HFSNets entwickelt und zur Verfügung gestellt. Der Einsatz von FSNets wird anhand von mehreren Beispielen aus aktuellen und praxisrelevanten Bereichen der verteilten Systeme demonstriert. Im einzelnen wird die Modellierung und Ausführung von *Geschäftsprozessen*, *elektronischen Verträgen* und *intelligenten Agenten* mit FSNets demonstriert.

Abstract

The increasing requirements on distributed software systems like internet applications have raised the complexity to a level which current approaches can no longer cope with. This trend leads to a high demand for approaches that allow an adequate *specification of distributed systems*. One of the corner stones of system specification is a modeling language or *modeling technique*, which defines syntax and semantics of models and provides *tools* for creating, analyzing, and validating models.

Feature-Structure-Nets (FSNets) are presented as a modeling technique for distributed systems. FSNets are based on feature structures and Petri nets. Feature structures support *information-oriented modeling*, while Petri nets are applied to specify the model's *process-oriented* aspects. After giving a survey of information- and process-oriented modeling techniques – focussing on the *Unified Modeling Language* (UML) – syntax and semantics of FSNets are formally defined. A UML notation for feature structures is developed and applied for FSNets. Thus, FSNets are the first Petri net formalism using UML for modeling and depicting types and objects in a coherent way. FSNets can be executed as a result of their *operational semantics*. Modeling with FSNets is supported by a graphical tool, which allows editing and execution of FSNets.

In the theoretical part, feature structures are studied as a formal notation for object graphs or *object views*. *Basic FSNets* and in particular *elementary FSNets* (EFSNets) are formally defined and used to gain insights into specific phenomena of distributed systems. The main contributions are defining *colored processes* of EFSNets as an extended process semantics of Nets in Nets and investigating consistency of distributed copies of objects or processes, taking into account the difference between *value semantics* and *reference semantics*.

As a practical result, basic FSNets are combined with concepts of Reference nets to define the class of higher FSNets (HFSNets). This step enhances the applicability of FSNets to real-world problems. A graphical, UML-based notation and a software tool for editing and executing HFSNets have been developed and deployed. Application of FSNets is demonstrated using examples from several up-to-date areas of distributed systems which are relevant in practice. The chosen applications are modeling and executing *business processes*, *electronic contracts*, and *intelligent agents* with FSNets.

Danksagung

Diese Arbeit entstand während meiner Tätigkeit als Mitarbeiter im Arbeitsbereich *Theoretische Grundlagen der Informatik* (TGI) und in der Arbeitsgruppe *Verteilte Systeme* (VSYS) am Fachbereich Informatik der Universität Hamburg.

Für die Betreuung der Promotion danke ich Prof. Dr. Winfried Lamersdorf und Prof. Dr. Rüdiger Valk, die in den vergangenen Jahren meine Arbeit durch konstruktive Kritik, hilfreiche Gespräche und mit großem Interesse begleitet haben. Nachdem ich bei Prof. Valk die theoretischen Grundlagen vor allem der Petrinetze erarbeiten konnte, gab mir Prof. Lamersdorf die Gelegenheit, dieses Wissen praxisorientiert umzusetzen. In beiden Arbeitsgruppen habe ich gleichermaßen die gute Atmosphäre genossen, die dem fachlichen und menschlichen Engagement der Leiter und Kollegen zu verdanken ist.

Prof. Dr. Christopher Habel verdanke ich ein hochinteressantes Vertiefungsgebiet während meiner Studienzeit, in der mein fortwährendes Interesse für Wissensrepräsentationstechniken und Logik geprägt wurde.

Die vorliegende Arbeit wäre ohne die Unterstützung vieler weiterer Personen nicht zu dem geworden, was sie ist. Besonders bedanken möchte ich mich bei Olaf Kummer, Daniel Moldt, Axel Wienberg und Marko Boger, die durch zahlreiche Diskussionen meine Sicht der Informatik entscheidend beeinflußt haben. In der letzten Phase haben vor allem Olaf Kummer, Berndt Farwer, Axel Wienberg und Daniel Moldt die Fertigstellung der Arbeit durch Anregungen, Kritik und tatkräftige Unterstützung vorangetrieben. Für die gute Zusammenarbeit in unserem gemeinsamen Projekt Renew möchte ich an dieser Stelle nochmals Olaf Kummer danken. Weiter gebührt mein Dank meinen Kollegen bei VSYS, TGI und im Cosmos-Projekt sowie den Studenten, die durch Studien- und Diplomarbeiten in angrenzenden Themengebieten indirekt zu dieser Arbeit beigetragen haben.

Einen großen Dank verdienen nicht zuletzt meine Eltern Ursula und Ernst, die mir das Studium ermöglicht haben, und Ulrike, die mir in schweren Zeiten zur Seite stand.

Inhaltsverzeichnis

1 Einleitung und Motivation **1**
- 1.1 Auszeichnende Eigenschaften verteilter Systeme . . . 2
  - 1.1.1 Konsistenz verteilter Information . . . 4
  - 1.1.2 Nebenläufigkeit und Synchronisation verteilter Prozesse . . . 10
- 1.2 Modellierung verteilter Systeme . . . 14
- 1.3 Aufbau der Arbeit . . . 21

2 Informationsorientierte Modellierung **23**
- 2.1 Klassen und Typen . . . 24
  - 2.1.1 UML-Konzepte und -Notationen . . . 25
  - 2.1.2 Formalisierung von Typen und Subsumption . . . 33
  - 2.1.3 Typsysteme aus Konzeptsystemen . . . 35
  - 2.1.4 Eine UML-Notation für Konzeptsysteme . . . 41
- 2.2 Objekte . . . 44
  - 2.2.1 UML-Konzepte und -Notationen . . . 45
  - 2.2.2 Objektsichten in UML . . . 48
  - 2.2.3 Verteilung in UML . . . 50
- 2.3 Feature Structures . . . 53
  - 2.3.1 Einführung . . . 54
  - 2.3.2 Formale Definition . . . 57
  - 2.3.3 Subsumption und Unifikation . . . 67
  - 2.3.4 Wohltypisierung . . . 80
  - 2.3.5 Erweiterte Notationen . . . 86
- 2.4 Implementierung einer Feature-Structure-Bibliothek . . . 93
  - 2.4.1 Objektorientiertes Grundmodell . . . 93
  - 2.4.2 Integration von Feature Structures und Java . . . 97
  - 2.4.3 Unifikationsalgorithmus . . . 104

3 Prozeßorientierte Modellierung **109**
- 3.1 Konzepte und Notationen . . . 111
  - 3.1.1 Modellierung von Dynamik in UML . . . 111

3.1.2 Zustandsdiagramme in UML . . . 112
3.1.3 Aktivitätsdiagramme in UML . . . 117
3.1.4 Aktivitätenmodelle . . . 124
3.1.5 Ereignisgesteuerte Prozeßketten . . . 125
3.2 Klassische Petrinetze . . . 128
3.2.1 Basis-Petrinetze . . . 128
3.2.2 S/T-Netze . . . 130
3.2.3 Höhere Petrinetze . . . 138
3.2.4 Erweiterte Kantenarten . . . 141
3.3 Netze in Netzen . . . 142
3.3.1 Elementare Objektsysteme . . . 144
3.3.2 Linearlogische Petrinetze (LLPN) . . . 148
3.3.3 Referenznetze . . . 152
3.3.4 Diskussion: Wert- versus Referenzsemantik . . . 155
3.4 Geschäftsprozeßmodellierung . . . 158
3.4.1 Einführung . . . 158
3.4.2 Höhere Petrinetze . . . 162
3.4.3 Workflow-Netze . . . 163
3.4.4 Prolog-Netze . . . 164
3.4.5 SGML- und XML-Netze . . . 166
3.5 Werkzeugunterstützung für Referenznetze: Renew . . . 169
3.5.1 Überblick . . . 170
3.5.2 Workflow-Erweiterungen . . . 170

4 Feature-Structure-Netze 173
4.1 Basis-Feature-Structure-Netze . . . 174
4.1.1 Formale Definition . . . 175
4.1.2 Graphische Notation . . . 176
4.1.3 Schaltfolgensemantik . . . 184
4.2 Elementare Feature-Structure-Netze . . . 186
4.2.1 Motivation: Wert- versus Referenzsemantik . . . 187
4.2.2 Formale Definition . . . 189
4.2.3 Schaltfolgen nach Wert- und Referenzsemantik . . . 192
4.2.4 Universelle elementare FSNets . . . 203
4.2.5 Kombination von Wert- und Referenzsemantik . . . 212
4.2.6 Schrittsemantik . . . 214
4.2.7 Erreichbare Markierungen . . . 215
4.2.8 Prozeßsemantik . . . 218
4.3 Netze in Netzen als FSNets . . . 236
4.3.1 System- und Objektnetz als ein FSNet . . . 236
4.3.2 Feature-Structure-Prozesse . . . 239
4.3.3 Diskussion . . . 245

4.4 Höhere Feature-Structure-Netze . . . 248
4.4.1 Netzexemplare und synchrone Kanäle . . . 249
4.4.2 Test-, Reservierungs- und Inhibitorkanten in HFSNets . . . 251
4.5 Werkzeugunterstützung: FSRenew . . . 252
4.6 Diskussion . . . 253

5 Anwendungsbeispiele 257
5.1 Geschäftsprozeßmodellierung . . . 257
5.1.1 Justizbehörde . . . 259
5.1.2 Online-Bestellungsverarbeitung . . . 264
5.2 E-Commerce: elektronische Verträge . . . 274
5.2.1 Cosmos: Eine Infrastruktur für elektronische Dienstemärkte . 275
5.2.2 Ausführung elektronischer Verträge in Cosmos . . . 281
5.2.3 Modellierung von Vertrags-Workflows mit FSNets . . . 285
5.3 Intelligente Agenten . . . 290
5.3.1 Software-Agenten, Actors und verteilte Künstliche Intelligenz . 290
5.3.2 BDI-Agenten . . . 292
5.3.3 Generische BDI-Agenten-Architektur . . . 293
5.3.4 Anwendungsbeispiel: Abfallsammler . . . 299

6 Zusammenfassung und Ausblick 309
6.1 Zusammenfassung . . . 309
6.2 Ausblick . . . 313

A Grundlegende Definitionen 317

B Definitionen zu Petrinetzen 321

C Konstruktion eines Feature-Structure-Prozesses 325

Literaturverzeichnis 331

Abbildungsverzeichnis

1.1 Ein Petrinetzmodell eines Client/Server-Modells mit pessimistischer Konsistenzkontrolle. 5
1.2 Ein Petrinetzmodell eines Client/Server-Modells mit optimistischer Konsistenzkontrolle. 6
1.3 Das Typsystem `OptimisticFSNetTypes` als Beispiel für ein UML-Modell zur Spezifikation von Typen für Objektmarken in Petrinetzen. 7
1.4 Das Client/Server-Modell mit optimistischer Konsistenzkontrolle als FSNet. 8
1.5 Unifikation modifizierter Daten in mehreren Schritten. 9
1.6 Drei Muster für den Start einer Interaktion nach [Engels et al. 2000], S. 311. 13
1.7 Drei Beispiele für vollständige Interaktionsmuster nach [Engels et al. 2000], S. 313. 14

2.1 Darstellung eines Typs in UML. 27
2.2 Verschiedene Arten von Subtypen in UML-Notation. 28
2.3 Darstellung einer Assoziation in UML-Notation. 30
2.4 Ein Ausschnitt des statischen Modells des *Abstract Windowing Toolkit* von Java als UML-Klassendiagramm. 31
2.5 Mehrfach-indirekte Inkonsistenz durch Feature-Typisierung. 39
2.6 Ein Konzeptsystem zur Modellierung der Struktur natürlichsprachlicher Sätze in UML-Notation. 42
2.7 Das Konzeptsystem aus Abbildung 2.6, wobei Attribute als Assoziationen dargestellt sind. 43
2.8 Darstellung von Objekten in UML-Notation. 46
2.9 Darstellung von Objektbeziehungen in UML-Notation. 47
2.10 Alternative Darstellung von Aggregaten in UML-Notation. 48
2.11 Darstellung von Aggregaten in der hier verwendeten Variante der UML-Notation. 49
2.12 Eine UML-Kollaboration auf Spezifikationsebene aus [Hitz und Kappel 1999], S. 125. 49

2.13 Ein UML-Verteilungsdiagramm, das die grobe Architektur der Renew-Workflow-Erweiterungen modelliert. 51
2.14 Modellierung von Objektmigration mit einem Verteilungsdiagramm. 52
2.15 Eine Feature Structure über dem Typsystem aus Abbildung 2.7, die einen Satz darstellt. 55
2.16 Skizze zum Beweis von Satz 2.23. 63
2.17 Skizze zum Beweis von Satz 2.36. 75
2.18 Skizze zur Knotenbenennung in Pfadgleichungen nach Definition 2.38. 79
2.19 Eine Unifikation von wohltypisierten Feature Structures, deren Ergebnis nicht wohltypisierbar ist. 85
2.20 Ein verschachteltes Tupel als Feature Structure in ausführlicher und abkürzender Notation. 88
2.21 Das Listen-Typsystem Type_{L} mit ausführlichen Bezeichnern in (a) und Kurzbezeichnern in (b). 89
2.22 Eine Liste als Feature Structure mit ausführlichen und Kurzbezeichnern in AVM- und in Graphnotation. 89
2.23 Die Liste aus Abbildung 2.22 als AVM in abkürzender Notation. . . 90
2.24 Eine offene Liste als AVM in abkürzender Notation. 90
2.25 Erweiterung der Subsumptionsrelation von Basistypen in (a) auf Listentypen in (b) und (c). 92
2.26 Analyse-Klassendiagramm des Feature-Structure-Systems. 94
2.27 Entwurfs-Klassendiagramm der Feature-Structure-Bibliothek. . . . 96
2.28 Entwurfs-Klassendiagramm der verschiedenen Konzeptklassen. . . 100
2.29 Entwurfs-Klassendiagramm mit Einbindung von Java-Typen. . . . 102
2.30 Die Implementierung der Unifikation als Klasse EquivRelation mit Hilfsklassen. 105

3.1 Eine einfache graphischen Oberfläche für eine Aufgabenliste (*to-do list*) und das zugehörige Zustandsdiagramm. 116
3.2 Bestellung und Auslieferung eines Produkts als Aktivitätsdiagramm. 119
3.3 Ein Ausschnitt aus dem Aktivitätsdiagramm aus Abbildung 3.2 mit Objektflüssen. 121
3.4 Das Aktivitätsdiagramm aus Abbildung 3.2 mit Verantwortlichkeitsbereichen. 122
3.5 Das Aktivitätsdiagramm aus Abbildung 3.4 mit Akteuren statt Verantwortlichkeitsbereichen. 123
3.6 Aktivitätenmodell eines Kreditbearbeitungs-Workflow nach [Jablonski et al. 1997], S. 167. 125
3.7 Ereignisgesteuerte Prozeßkette des Kreditbearbeitungs-Workflow nach [Jablonski et al. 1997], S. 169. 126
3.8 Konnektivitätsgraph eines Auftragssystems aus [Jessen und Valk 1987], S. 25. 129

3.9 Dem Auftragssystem aus Abbildung 3.8 zugeordnetes Kausalnetz aus [Jessen und Valk 1987], S. 26. 129
3.10 Vergröberung bzw. Faltung eines B/E-Netzes zu einem S/T-Netz mit Kapazitäten und Kantengewichten aus [Jessen und Valk 1987], S. 35. 131
3.11 Beispiel für eine Kapazitätsbeschränkung, die zu einem Konflikt im Nachbereich führt. 131
3.12 Konstruktion einer Komplementärstelle zu einer Stelle mit Kapazitätsbeschränkung aus [Jessen und Valk 1987], S. 51. 132
3.13 Ein markiertes S/T-Netz ohne Kapazitäten aus [Jessen und Valk 1987], S. 35. 133
3.14 Wechselseitiger Ausschluß von Schreibern und Lesern mit zwei Aufträgen (in [Jessen und Valk 1987], S. 39, mit drei Aufträgen). 134
3.15 Der Erreichbarkeitsgraph des S/T-Netzes aus Abbildung 3.14. 135
3.16 Der Erreichbarkeitsgraph des S/T-Netzes aus Abbildung 3.14 mit zusätzlichen Kanten für Schritte. 136
3.17 Ein Prozeß des S/T-Netzes aus Abbildung 3.14 leicht modifiziert aus [Jessen und Valk 1987], S. 42. 137
3.18 Ein gefärbtes Petrinetz in Renew-Syntax aus [Kummer und Wienberg 2000], S. 22. 140
3.19 Ein elementares Objektsystem aus Objektnetz (ON, links) und Systemnetz (SN, rechts) aus [Valk 1999]. 145
3.20 Eine Prozeßmarkierung des Systemnetzes aus Abbildung 3.19 (aus [Valk 1999]). 146
3.21 Beispiele für die Schaltregel für EOS mit Prozeßmarkierungen (aus [Valk 1999], dort Abbildung 14): (a) Unifikation und Kopieren von Prozessen, (b) zusätzliche Interaktion und (c) objektautonomes Schalten. 147
3.22 S/T-Netz, das in eine linearlogische Formel übersetzt werden soll, erweitert aus [Farwer 1999], S. 226. 150

4.1 Das Konzeptsystem, das den folgenden FSNets zugrundeliegt. 176
4.2 Ein Beispiel für die erste Variante der Notation von FSNets. 177
4.3 Zweite Variante: In der Notation aus Abbildung 4.2 wurden Pfade durch Knotenreferenzen ersetzt. 178
4.4 Dritte Variante: In der Notation aus Abbildung 4.3 wurden von Kanten aus nicht erreichbare Knoten in Transitionsregeln weggelassen. . 179
4.5 Vierte Variante: In der Notation aus Abbildung 4.2 wurden die Transitionsregel-AVMs direkt an die entsprechenden Kanten geschrieben. 180
4.6 Fünfte Variante und Standard: Die Notationsmöglichkeiten aus Abbildung 4.4 und Abbildung 4.5 wurden kombiniert. 180

4.7 Das FSNet in Standardnotation wie in Abbildung 4.6, jedoch mit UML-Notation für Feature Structures und alphanumerischen Knotenreferenzen. . . . 181
4.8 Das FSNet aus Abbildung 4.2 mit Feature Structures in Graphnotation. . . . 182
4.9 Die Graphnotation aus Abbildung 4.8 in vereinfachter Notation. . . . 183
4.10 Ein elementares FSNet, das unter Verwendung von Referenzsemantik eine Liste umkehrt und nebenläufig ergänzt. . . . 188
4.11 Ein elementares FSNet, das unter Verwendung von Wertsemantik das Verhalten des Netzes aus Abbildung 4.10 nachbildet. . . . 189
4.12 Das Markierungs-Typsystem zum Typsystem aus Abbildung 4.1 und zur Stellenmenge $S = \{p1, p2, p3, p4, p5\}$. . . . 191
4.13 Das Transitionsregel-Typsystem zum Netz aus Abbildung 4.10. . . . 199
4.14 Abgeleitete Transitionsregel und Wirkung von `t1` aus dem Netz aus Abbildung 4.10. . . . 200
4.15 Die abgeleitete Transitionsregel von `t2` aus dem Netz aus Abbildung 4.10. . . . 202
4.16 Ein einfaches EFSNet mit seinem Typsystem in (a), dessen universelles EFSNet-Typsystem in (b) und die Kodierung des Netzes als Feature Structure über diesem Typsystem in (c). . . . 205
4.17 Anwendung einer Transitionswirkung auf eine Markierung als FSNet modelliert. . . . 206
4.18 Ein universelles EFSNet, das beliebige EFSNets nach Referenzsemantik ausführen kann. . . . 207
4.19 Ein universelles EFSNet, das beliebige EFSNets nach Wertsemantik ausführen kann. . . . 209
4.20 Ein universelles EFSNet, das beliebige EFSNets nach Wert- oder Referenzsemantik ausführen kann. . . . 211
4.21 Ein Petrinetz, dessen Stellen in Partitionen unterteilt sind. . . . 213
4.22 Die Wirkung des Schritts $\{t2, t4\}$ aus dem Netz aus Abbildung 4.10. 216
4.23 Der Erreichbarkeitsgraph des Netzes aus Abbildung 4.10 nach Referenzsemantik. . . . 217
4.24 Der Erreichbarkeitsgraph des Netzes aus Abbildung 4.11 nach Wertsemantik. . . . 219
4.25 Das Prozeß-Typsystem zu dem Netz aus Abbildung 4.10. . . . 224
4.26 Ein EFSNet-Prozeß nach Referenzsemantik des Netzes aus Abbildung 4.11 (grau) und sein maximaler S-Schnitt (schwarz). . . . 228
4.27 Die Prozeß-Transitionsregel $r_p(t1)$ der Transition t1 aus Abbildung 4.10 230
4.28 Die einzelnen Schritte der Erstellung eines Referenzsemantik-Prozesses des Netzes aus Abbildung 4.10. . . . 233
4.29 Die einzelnen Schritte der Erstellung eines Referenzsemantik-Prozesses des Netzes aus Abbildung 4.10 (Fortsetzung). . . . 234

4.30 Eine auf kausale Abhängigkeiten reduzierte Sicht des Prozesses P_5 aus den Abbildungen C.6 und 4.29. 235
4.31 Ein Wertsemantik-Prozeß des Netzes aus Abbildung 4.11. 235
4.32 Das EOS aus Abbildung 3.19 als FSNet. 238
4.33 Das Prozeß-Typsystem des EOS aus Abbildung 3.19. 241
4.34 Feature-Structure-Prozesse des EOS aus Abbildung 3.19 über dem Typsystem aus Abbildung 4.33 als AVMs: (1) initialer Prozeß, (2) `t8` und `e5` schalten autonom, (3) `t1` und `e1` schalten synchron. 243
4.35 Feature-Structure-Prozeß des EOS aus Abbildung 3.19 über dem Typsystem aus Abbildung 4.33. 246

5.1 Ein einfaches generisches Typsystem zur Workflow-Modellierung. . . 259
5.2 Das Typmodell für den Geschäftsprozeß des niederländischen Gerichts. 260
5.3 Der Gerichts-Geschäftsprozeß als HFSNet. 262
5.4 Ein FSNet zur Simulation einer Workflow-Umgebung für den Geschäftsprozeß aus Abbildung 5.3. 263
5.5 Das Konzeptsystem für die vereinfachte Online-Bestellungsverarbeitung. 266
5.6 Die vereinfachte Online-Bestellungsverarbeitung als FSNet modelliert. 266
5.7 Das Konzeptsystem für die Online-Verarbeitung von Bestellungen mit mehrerer Produkten. 268
5.8 Die Online-Verarbeitung von Bestellungen mit mehrerer Produkten als FSNet modelliert. 270
5.9 Die Erweiterung des Konzeptsystems der Online-Bestellungsverarbeitung aus Abbildung 5.7 um Konzepte für synchrone Kanäle. 271
5.10 Die Online-Bestellungsverarbeitung aus Abbildung 5.8 als HFSNet. . 272
5.11 Das Lager zur Online-Bestellungsverarbeitung aus Abbildung 5.10 als eigenes Netzobjekt modelliert. 273
5.12 Die drei Phasen einer Geschäftstransaktion in COSMOS. 276
5.13 Komponenten von COSMOS. 278
5.14 Das objektorientierte Vertragsmodell von COSMOS (vereinfacht). . . 280
5.15 Architektur der Vertragsausführungs-Komponente von COSMOS. . . 282
5.16 Abstraktes Entwurfsmodell der Workflow-Definition des COSMOS-Vertragsmodells aus Abbildung 5.14. 283
5.17 Auslieferung und Bezahlung eines Produkts als Vertrags-Workflow. . 286
5.18 Das aus dem Vertrags-Workflow in Abbildung 5.17 abgeleitete Typsystem. 287
5.19 Vertrags-Workflow zur Auslieferung und Bezahlung eines Produkts als FSNet. 288

5.20 Die Architektur eines BDI-Agenten als schematisches FSNet. 294
5.21 Das Typmodell zur BDI-Agenten-Architektur. 295
5.22 Das verfeinerte BDI-Agenten-Modell als HFSNet BDIAgent. 297
5.23 Die Wissensbasis eines BDI-Agenten als HFSNet KB. 298
5.24 Die Erweiterung des BDI-Agenten-Typsystems für einen konkreten Agenten, der Abfall sammeln soll. 301
5.25 Das initiale Wissen des Abfallsammler-Agenten. 302
5.26 Die initialen Pläne des Abfallsammler-Agenten. 303
5.27 Ein konkreter BDI-Agent, der Abfall sammeln kann, als HFSNet *WasteCollector*. 305
5.28 Eine Welt mit einem Abfallsammler und zwei Abfall-Ereignissen. . . 307
5.29 Die beiden Intentionen, die durch die in Abbildung 5.28 erzeugten Ereignisse in der Stelle Intentions im Netz BDIAgent generiert werden. 307

C.1 Der initiale Prozeß P_0 des elementaren FSNets aus Abbildung 4.10 als AVM. 325
C.2 Der maximale S-Schnitt scut(P_0) ergibt mit der Prozeß-Transitionsregel von t1 den Prozeß P_1. 326
C.3 Der maximale S-Schnitt scut(P_1) ergibt mit der Prozeß-Transitionsregel von t2 den Prozeß P_2. 327
C.4 Der maximale S-Schnitt scut(P_2) ergibt mit der Prozeß-Transitionsregel von t4 den Prozeß P_3. 328
C.5 Der maximale S-Schnitt scut(P_3) ergibt mit der Prozeß-Transitionsregel von t2 den Prozeß P_4. 329
C.6 Der maximale S-Schnitt scut(P_5) ergibt mit der Prozeß-Transitionsregel von t3 den Prozeß P_5. 330

Tabellenverzeichnis

1.1 Informations- und prozeßorientierte Modellierungssichten und die in dieser Arbeit verwendeten Techniken. 17

2.1 Zusammenhang zwischen Assoziationen und Operationen der Klassen aus Abbildung 2.26 und den formalen Definitionen aus den Abschnitten 2.1 und 2.3. 97

Kapitel 1

Einleitung und Motivation

Diese Arbeit schlägt eine neue Modellierungstechnik für verteilte Systeme vor. Um eine praxisrelevante, anwendbare, verständliche und dennoch mächtige Modellierungssprache zu erhalten, ist es unumgänglich, die Probleme zu analysieren, die in heutigen verteilten Systemen auftreten.

Durch die globale Tendenz der Vernetzung, das Internet und Intranets innerhalb von Organisationen gibt es inzwischen kaum noch ein Anwendungssystem, das ohne Verteilung auskommt. Nachdem sich in den 80er und 90er Jahren die klassische Software-Entwicklung immer weiter zur Ingenieurswissenschaft entwickelt hat und durch die Unified Modeling Language (UML) nun über eine standardisierte Modellierungstechnik verfügt, steht man bei der Entwicklung verteilter Systeme noch vor großen Problemen. Die Verteilungsaspekte sind zu den eigentlichen Herausforderungen bei Analyse, Entwurf und Implementierung von Anwendungssystemen geworden.

Will man sich diesen Problemen stellen, so gilt es, gemeinsame Phänomene in den vielfältigen Anwendungsgebieten verteilter Systeme zu identifizieren und als generelle Konzepte vereinheitlicht darzustellen. Im Bereich der verteilten Systeme wurden bereits viele derartige Konzepte identifiziert, die aber noch keine einheitliche Modellierungstechnik hervorgebracht haben. Für viele Konzepte wie Prozesse existieren mathematische Abstraktionen, die für den Praktiker schwer anzuwenden sind. Andererseits gibt es in der Praxis viele nützliche Konzepte wie den „Agenten“, die kaum formal beschrieben werden können. Zwischen Theorie und Praxis klafft also noch eine große Lücke, die von einer formal definierten *und* praktisch anwendbaren Modellierungstechnik überbrückt werden muß.

In dieser Arbeit wird eine konkrete Modellierungstechnik aus der Praxis motiviert, formal definiert, durch ein Software-Werkzeug unterstützt und in verschiedenen Anwendungsgebieten beispielhaft erprobt. Dazu wurde der folgende Aufbau der Arbeit gewählt: Es soll sich der Theorie aus der Praxis genähert werden, um die theoretischen Teile der Arbeit in einen praktischen Kontext zu stellen. Dazu werden verschiedene Modellierungstechniken betrachtet, wobei nach informations-

und prozeßorientierten Techniken unterteilt vorgegangen wird. Für beide Bereiche wird, ausgehend von Anforderungen der verteilten Systeme, auf bestehende theoretische Grundlagen zurückgegriffen, die durch Auswahl und Kombination schließlich eine neue Modellierungstechnik formen. Für die informationsorientierte Modellierung werden Feature Structures und für die prozeßorientierte Modellierung Petrinetze herangezogen und zu einer gemeinsamen Technik verbunden. In der konkreten Ausgestaltung der Modellierungstechnik, *Feature-Structure-Netze* oder kurz *FSNets* genannt, wird die aktuelle Forschung, vor allem aus den Bereichen Petrinetze und Objektorientierung, berücksichtigt. Anhand von eingeschränkten, sogenannten *elementaren* FSNets (*EFSNets*) können weitergehende Formalisierungen vorgenommen werden, die zu einem besseren theoretischen Verständnis von FSNets und zu einer neuen Sicht auf höhere Petrinetze, vor allem Netze in Netzen, beitragen. Um die Klammer zu schließen, wird die Modellierungstechnik durch Implementierung eines geeigneten Werkzeugs unterstützt und im letzten Teil der Arbeit zur Modellierung einiger ausgewählter Probleme aus Anwendungsgebieten verteilter Systeme eingesetzt.

Diese Einleitung fährt fort mit einem Überblick über die auszeichnenden Eigenschaften verteilter Systeme (Abschnitt 1.1), die anhand von Beispielen aus den Bereichen Informations- und Prozeßmodellierung motiviert werden. In Abschnitt 1.2 werden allgemeine Aspekte der Modellierung sowie der Einfluß der besonderen Eigenschaften verteilter Systeme auf deren Modellierung und damit auf eine entsprechende Modellierungstechnik behandelt. Die Einleitung schließt mit einem Überblick über die einzelnen Teile der Arbeit (Abschnitt 1.3).

1.1 Auszeichnende Eigenschaften verteilter Systeme

Heutzutage zählt ein breites Spektrum von Anwendungen zum Gebiet der verteilten Systeme. Im weitesten Sinne kann man jedes Anwendungssystem als verteiltes System auffassen, das ein Netzwerk benutzt und damit eine physikalische Verteilung der Hardware-Komponenten impliziert.

Allein durch die ständig wachsende Zahl der Internet-Anwendungen gibt es einen großen Bedarf an einer Modellierungstechnik für verteilte Anwendungssysteme. Aber auch Workflow-Management-Systeme, Agenten-Systeme und Komponententechnologien basieren zu einem großen Teil auf den Konzepten und Techniken aus dem Bereich der verteilten Systeme. Das verteilte System ist also bereits die erste wichtige Abstraktion, die es zu verstehen gilt, um eine angemessene Modellierungstechnik zu entwerfen.

Der Begriff des *verteilten Systems* wird in [Lamersdorf 1994] anhand der Eigenschaften solcher Systeme eingeführt. Demnach sind verteilte Systeme moderne verteilte Rechnerarchitekturen, die sich durch die folgenden Charakteristika von klassischen, zentralistischen Rechensystemen abgrenzen:

- Es gibt viele unabhängige und heterogene Prozessoren, Speichersysteme, Subsysteme etc.,
- die Kommunikationsmöglichkeiten sind vielfältiger und die Nachrichtenlaufzeiten durch beschränkte Bandbreite und hohe Latenzzeiten nicht mehr vernachlässigbar,
- das Gesamtsystem zeigt ein von der Dysfunktion von Teilkomponenten möglichst unabhängiges Fehlerverhalten und
- es existiert ein zumindest minimaler gemeinsamer Systemzustand.

Aus der Existenz vieler Prozessoren folgt, daß verteilte Systeme in hohem Maße *nebenläufig* arbeiten. Mehrere Speichersysteme führen dazu, daß Daten im System nicht immer direkt referenzierbar sind, sondern zum Teil nur durch *Replikation* als Kopie zur Verfügung stehen. Die Kommunikationsmöglichkeiten sind zwar vielfältiger, lassen sich aber erst durch die Integration verschiedener Teilnetze und Protokolle nutzen ([Lamersdorf 1994]). Die Redundanz von Teilkomponenten führt zu einer erhöhten Zuverlässigkeit und Verfügbarkeit, dies aber auf Kosten einer höheren Komplexität.

Als letztes Charakteristikum wird von Lamersdorf ein minimaler gemeinsamer Systemzustand gefordert. Dieser gemeinsame Systemzustand muß allerdings nur konzeptionell vorhanden sein, nicht physikalisch. Die Komponenten oder Funktionseinheiten eines verteilten Systems können beispielsweise üblicherweise nicht auf gemeinsamen Speicher zugreifen, wohl aber durch Kommunikation mit einem geeigneten Protokoll bestimmte Eigenschaften ihres Systemzustands anderen Funktionseinheiten zusichern. Jessen und Valk definieren ein verteiltes System als „ein System von Funktionseinheiten, dessen globaler Zustand von keinem Algorithmus oder Protokoll benutzt wird“ ([Jessen und Valk 1987], S. 249). Dennoch ist es sinnvoll, einen minimalen gemeinsamen Systemzustand zu fordern, um bei der Menge von Funktionseinheiten überhaupt von *einem System* sprechen zu können.

Lamersdorf nennt in [Lamersdorf 1994] als Vorteile verteilter Systeme

- deren durch Zusammenschluß vieler einzelner Ressourcen und Dienste erweiterte Fähigkeiten,
- die Möglichkeit, die organisatorische Gliederung eines Unternehmens direkt in der Rechnerarchitektur widerzuspiegeln,
- die stärkere Modularisierung des Gesamtsystems und die leichte Erweiterbarkeit (*scalability*),
- die Kostenvorteile vieler kleiner Komponenten im Gegensatz zu einer Hochleistungskomponente und schließlich

- die erhöhte Sicherheit beispielsweise durch Vergabe unterschiedlicher Benutzerrechte.

Um diese Vorteile jedoch ausnutzen und umsetzen zu können, müssen eine Reihe von Problemen bewältigt werden. Die Probleme präsentieren sich als Kehrseite der Medaille der genannten Vorteile: Die Heterogenität und Komplexität des verteilten Systems muß beherrscht werden. Dies betrifft die Vereinheitlichung von Kommunikationsunterstützung, den Umgang mit partiellem Fehlerverhalten und sich zwischen Systemteilen propagierenden Fehlern sowie die Vermeidung von Engpässen (*bottlenecks*), welche die Vorteile des verteilten Systems zunichte machen können.

Konkreter wird die Realisierung verteilter Systeme heutzutage durch verschiedene Techniken und Werkzeuge vorangetrieben, welche eine verteilte Anwendungsprogrammierung erlauben. Ein bekannter Spezialfall ist die Entwicklung von Internet-Anwendungen, hier vor allem im Bereich des E-Commerce. Während die Implementierung von verteilten Systemen inzwischen durch *Middleware*, Programmiersprachen und Werkzeuge immer besser unterstützt wird, steht der Bereich der Modellierung verteilter Systeme noch an seinem Anfang. Die Modellierung verteilter Systeme wird, da sie einen Schwerpunkt dieser Arbeit darstellt, in dieser Einleitung gesondert in Abschnitt 1.2 vertieft.

Im Rahmen dieser Arbeit wird der Fokus vor allem auf zwei aus den Charakteristika verteilter Systeme resultierende Phänomene gesetzt. Eine die Information betreffende Eigenheit verteilter Systeme ist, daß kein referenzieller Zugriff auf entfernte Daten mehr möglich ist, sondern auf verteilten, zum Teil replizierten Daten gearbeitet wird. Die Prozeßsicht betrifft die Eigenschaft, daß in verteilten Systemen nicht von Synchronizität, sondern von asynchroner Kommunikation und Nebenläufigkeit ausgegangen werden muß. Beide Aspekte sollen in den folgenden Abschnitten anhand motivierender Beispiele verdeutlicht werden.

1.1.1 Konsistenz verteilter Information

Die Verteilung jeglicher veränderlicher Information, sei es durch Datenbanken, Datenfernübertragung oder andere Verfahren, ist eine grundlegende Problemstellung verteilter Systeme. Durch die örtliche Verteilung von Daten kann auf diese nicht mehr synchron zugegriffen werden, so daß es spezielle Algorithmen erfordert, um beispielsweise einen global konsistenten Zustand (*global snapshot*) sämtlicher Daten zu erzeugen ([Mattern 1989]).

Das Phänomen der Konsistenz von Daten soll hier anhand des aus der Literatur bekannten *Leser-Schreiber-Problems* ([Jessen und Valk 1987]) insbesondere für verteilte Systeme motiviert werden. Das Leser-Schreiber-Problem wird üblicherweise durch einen *wechselseitigen Ausschluß* von lesenden und schreibenden Prozessen gelöst. Dafür wird die Veränderung von zentral gespeicherten Daten durch einen *kritischen Abschnitt* geschützt, den nur ein Schreiber zur Zeit betreten darf und auch nur dann, wenn kein Prozeß mehr lesend auf die Daten zugreift.

Um den verteilten Charakter dieses Problems deutlich zu machen, gehen wir von einer Client/Server-Architektur aus. Klienten können Lese- oder Schreibwünsche äußern, die vom Server beantwortet werden. Der Server ist für die Konsistenz der Daten verantwortlich.

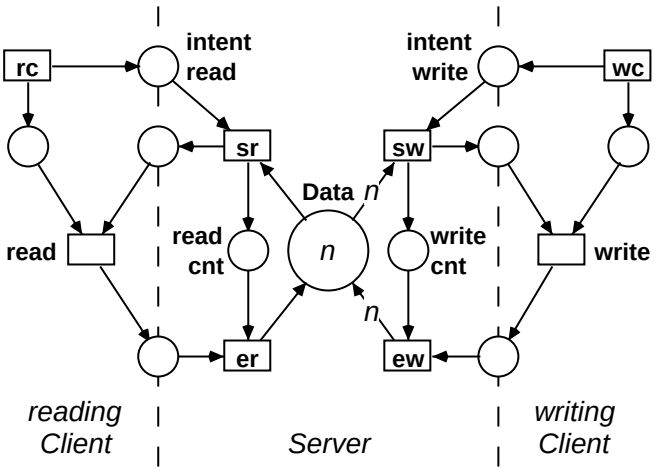

Abbildung 1.1: Ein Petrinetzmodell eines Client/Server-Modells mit pessimistischer Konsistenzkontrolle.

Das in Abbildung 1.1 dargestellte Petrinetz[1] modelliert die Vergabe von Lese- und Schreibrechten vom Server (mittlerer Bereich) an lesende (links) oder schreibende (rechts) Klienten. Dabei wird angenommen, daß die Netzelemente im jeweiligen Bereich Teile der jeweiligen Komponente (Client bzw. Server) modellieren. Die Stellen, die auf einer der gestrichelten Linien zwischen zwei Bereichen liegen, dienen als *Kommunikationskanäle* zwischen den Komponenten.

Die Stelle Data repräsentiert die zentrale Datenhaltung und enthält anfangs n Marken. Jede Marke stellt eine Leseerlaubnis dar, die der Server einem Klienten mit Lesewunsch ausgeben kann. Der Lesewunsch eines Klienten wird durch das Schalten der Transition rc (*reading client*) simuliert, die dies durch markieren des Kommunikationskanals intent read an den Server übermittelt und gleichzeitig in der lokalen, unbenannten Stelle vermerkt. Der Server kann nun durch Schalten der Transition sr (*start read*) eine Leseerlaubnis ausgeben und vermerkt jede ausgegebene Leseerlaubnis durch eine Marke in der Stelle read cnt. Ist der Lesevorgang beendet (Transition read), wird dies durch den unbenannten Kanal unten links an den Server übermittelt, der durch die Transition er (*end read*) die Leseerlaubnis aus der Stelle read cnt zurück in die Stelle Data legt und so wieder für andere Klienten zur Verfügung stellen kann.

[1] Petrinetze werden in Abschnitt 3.2 ausführlich vorgestellt.

Analog läuft die Ausgabe einer Schreiberlaubnis ab, indem erst wc (*writing client*) und sw (*start write*) schalten, nur müssen für letzteres *sämtliche* Leseerlaubnis-Marken in der Stelle Data vorhanden sein (Anschrift n an den Kanten nach sw und von ew). Die Ausgabe einer Schreiberlaubnis wird in der Stelle write cnt gespeichert. Entsprechend werden nach Beendigung des Schreibvorgangs (Transition write) vom Server n Marken zurückgelegt.

Diese Lösung stellt einen klassischen Sperrmechanismus (*locking*) dar, der zwar für eine zentralisierte Architektur unerwünschtes Verhalten ausschließt, aber erstens wenig Nebenläufigkeit erlaubt und zweitens nicht ohne weiteres auf eine Architektur ohne zentralen Server zu übertragen ist. Für den ersten Fall stelle man sich vor, daß mehrere schreibende Klienten nur unterschiedliche Bereiche der Daten ändern möchten. Trotzdem wird hier nicht erlaubt, daß diese die Daten gleichzeitig bearbeiten. Auch werden in der Praxis verteilte Datenbanken oft so konstruiert, daß sie während der Ausgabe einer Schreiberlaubnis lesenden Klienten eine Kopie der alten Daten liefern ([Date 1995]). Die üblicherweise gewählte Lösung, um die Nebenläufigkeit zu steigern, ist, die Daten in kleinere Einheiten zu unterteilen, die separat gesperrt werden können ([Heuer 1997]). Der zweite Fall, die Übertragung auf eine dezentrale Architektur, erfordert, daß jeder Knoten als Server für die von ihm lokal gehaltenen Daten agiert, da es in einer Lösung mit Sperrmechanismus immer nur einen Prozeß geben darf, der den Zugriff auf einen Datensatz verwaltet.

Weil diese erste Lösung davon ausgeht, daß bei der gleichzeitigen Bearbeitung durch mehrere Klienten inkonsistente Daten entstehen, wird diese hier als *pessimistische Konsistenzkontrolle* bezeichnet.

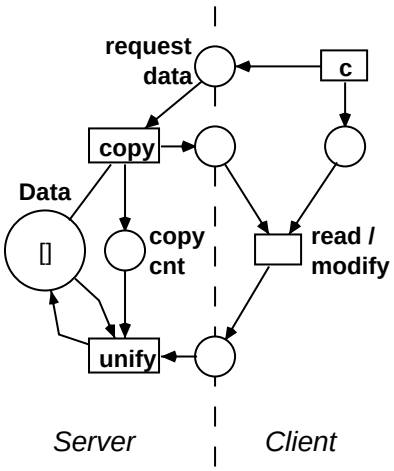

Abbildung 1.2: Ein Petrinetzmodell eines Client/Server-Modells mit optimistischer Konsistenzkontrolle.

Bei einer *optimistischen Konsistenzkontrolle* geht man dagegen davon aus, daß im Normalfall Klienten Datensätze so modifizieren oder besser ergänzen, daß sich auch bei nebenläufiger Bearbeitung ein konsistenter Zustand ergibt. Weiterhin findet in vielen Fällen kaum nebenläufige Bearbeitung der gleichen Datensätze statt. Da hier Inkonsistenzen als Ausnahmefall betrachtet werden, wird die Konsistenz zu den Originaldaten erst geprüft, wenn der Klient die eventuell modifizierten Daten zurückliefert.

Abbildung 1.2 zeigt ein Petrinetz, das ein Client/Server-Modell mit einer optimistischen Konsistenzkontrolle darstellt. Hier wird nicht nach Lese- oder Schreibwunsch unterschieden, so daß es nur eine Art von Klienten gibt. Der Server liefert auf Anforderung (Stelle request data) eine Kopie der Daten an den Klienten. In der Stelle copy cnt legt der Server für jede ausgegebene Kopie eine Marke ab. Der Klient hat die freie Wahl, ob er Daten nur lesen oder auch modifizieren möchte (Transition read / modify). Die eventuell modifizierten Daten werden zurück an den Server geschickt, der in der Transition unify versucht, die lokal vorhandenen und die zurückgelieferten Daten abzugleichen. Der Ausnahmefall des Scheiterns dieses Abgleichs ist im Modell nicht weiter spezifiziert.

Bisher wurde zwar über Daten gesprochen, diese werden aber in dem Petrinetz aus Abbildung 1.2 nicht weiter modelliert. An dieser Stelle soll als Vorausgriff eine Motivation für die in dieser Arbeit vorgeschlagene Modellierungstechnik, *Feature-Structure-Netze* oder kurz FSNets, gegeben werden. FSNets werden in Kapitel 4 definiert und untersucht. In Kapitel 5 wird an verschiedenen Beispielen Modellierung mit FSNets demonstriert.

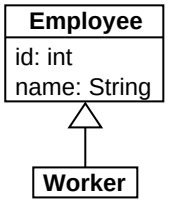

Abbildung 1.3: Das Typsystem `OptimisticFSNetTypes` als Beispiel für ein UML-Modell zur Spezifikation von Typen für Objektmarken in Petrinetzen.

In Abbildung 1.3 wird ein Beispiel für die hier vorgeschlagene Typmodellierung gezeigt. Typen werden in UML-Notation mit ihren typisierten Attributen und ihren Vererbungsbeziehungen angegeben. Kapitel 2 stellt diese Art der informationsorientierten Modellierung ausführlich vor.

Im Beispiel in Abbildung 1.3 werden zur Darstellung der Daten die Typen Employee (Angestellter) und Worker (Arbeiter) modelliert, wobei ein Arbeiter die Attribute id (eine Personalnummer) vom Typ int (ganze Zahl) und name (sein Name) vom Typ String (Zeichenkette) besitzt. Der Typ Worker spezialisiert Employee

und erbt somit dessen Attribute.

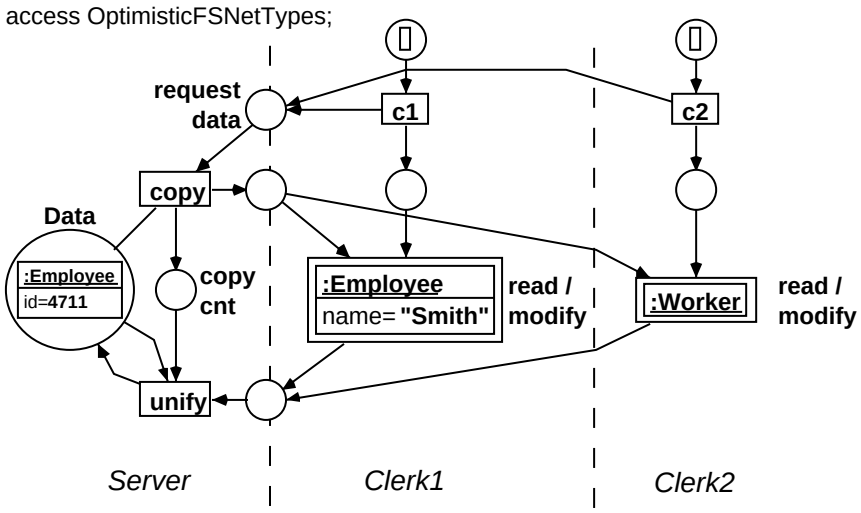

Abbildung 1.4: Das Client/Server-Modell mit optimistischer Konsistenzkontrolle mit konkreten Daten und Klienten als FSNet.

Ein FSNet besteht aus einem Typmodell und einem Petrinetz mit speziellen Marken und Netzanschriften. Abbildung 1.4 zeigt das Petrinetz-Modell aus Abbildung 1.2 als FSNet mit zwei konkreten Klienten, die bestimmte Modifikationen an den Daten vornehmen sollen. Das Netz nutzt das Typmodell aus Abbildung 1.3, was durch die Deklaration `access OptimisticFSNetTypes;` (links oben) gekennzeichnet wird.

Als Marken und Netzanschriften werden *Objekte* oder genauer gesagt *Objektsichten* in Form von *Feature Structures* angegeben, wobei hier wie bei der Modellierung des Typsystems auf eine UML-artige Notation zurückgegriffen werden kann[2]. Die Stelle Data enthält im Beispiel eine Objektsicht, die einen Angestellten mit der Personalnummer 4711 beschreibt.

Eine Transition eines FSNet *unifiziert* alle Objektsichten, die als Marken auf ihren Eingangsstellen liegen, miteinander und mit der *Transitionsregel* (dies ist die in der Transition notierte Objektsicht). Unifikation ist hier eine Operation auf Feature Structures, die zum oben genannten Abgleich modifizierter Daten eingesetzt werden kann. Auch die *Modifikation*, genauer gesagt nur die *Spezialisierung* von Daten kann durch Unifikation beschrieben werden.

[2]Für Feature Structures wird in der Literatur ([Carpenter 1992, Pollard und Sag 1994]) die in Abschnitt 2.3 beschriebene AVM-Notation verwendet. In FSNets können, je nach Wahl des Modellierers, beide Notationen alternativ eingesetzt werden. Die UML-Notation für Feature Structures ist ein Ergebnis dieser Arbeit und wird in Abschnitt 2.3.5 eingeführt.

Die im Beispiel gezeigten konkreten Klienten führen durch die angegebenen Transitionsregeln die folgenden Modifikationen durch. Der linke Klient fügt zu den Daten die Information hinzu, daß der Angestellte Smith heißt. Der rechte Klient ergänzt die Information, daß der Angestellte ein Arbeiter (Worker) ist. Die Transition unify führt nebenläufig vorgenommene Änderungen durch Unifikation zusammen.

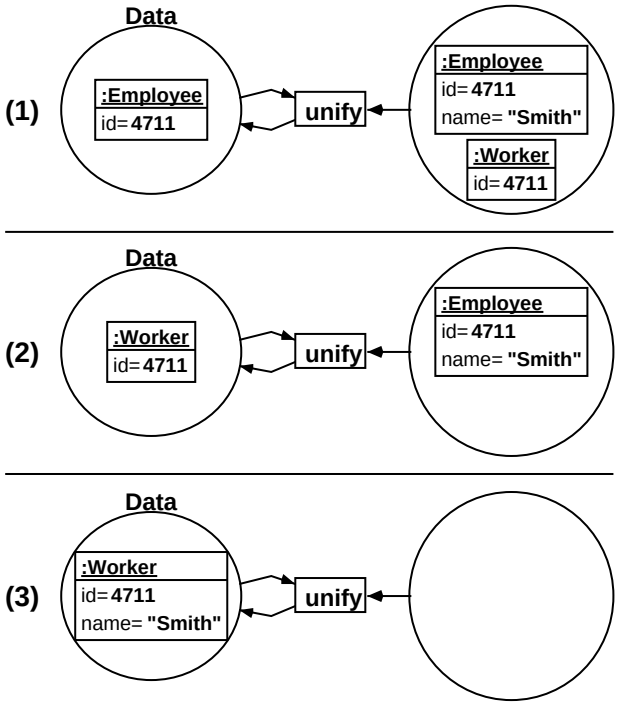

Abbildung 1.5: Unifikation modifizierter Daten in mehreren Schritten.

Der Vorgang der Unifikation der modifizierten Daten soll an diesem Beispiel verdeutlicht werden. Dazu zeigt Abbildung 1.5 einen Ausschnitt aus dem FSNet aus Abbildung 1.4 in mehreren aufeinanderfolgenden Markierungen. Nehmen wir an, daß beide Klienten ihre Modifikationen nebenläufig durchgeführt haben. Diese Situation ist in Abbildung 1.5 (1) zu sehen. Die beiden von den Klienten gelieferten modifizierten Kopien können nun in beliebiger Reihenfolge den Originaldaten hinzugefügt werden. Im Beispiel wird im ersten Schritt die Information des rechten Klienten hinzugefügt, was die in Abbildung 1.5 (2) gezeigte Markierung ergibt. Man beachte den spezialisierten Typ der Marke in der Stelle Data. Im zweiten Schritt wird die Information des linken Klienten hinzugefügt, was zur Markierung in Abbildung 1.5 (3) führt.

Die optimistische Konsistenzkontrolle bietet gegenüber der pessimistischen Variante vor allem in verteilten Systemen erhebliche Vorteile. Die oben erwähnten

Einschränkungen der nebenläufigen Bearbeitung von Daten bei der pessimistischen Kontrolle sind insofern nicht mehr gegeben, als die optimistische Kontrolle gleichzeitiges Lesen und Bearbeiten von Daten durch beliebig viele Klienten erlaubt.

Weiterhin ist es in vielen Situationen in einem verteilten System schlichtweg nicht möglich, wie bei der pessimistischen Kontrolle eine Schreiberlaubnis für Daten von einer zentralen Instanz zu erhalten, z.B. weil ein Klient zeitweilig keinen Kontakt zum Netzwerk hat (*off line*). In solchen Fällen eine Schreiberlaubnis über die gesamte *off line*-Zeit zu behalten, würde den nebenläufigen Zugriff auf Daten noch weiter reduzieren, als dies bei der pessimistischen Kontrolle ohnehin bereits der Fall ist. Die optimistische Kontrolle erlaubt dagegen die lokale Bearbeitung vieler Kopien durch verteilte Klienten, auch solche, die *off line* sind.

Neben diesen Vorteilen erzeugt der optimistische Ansatz allerdings eine hohe Komplexität. In dieser Arbeit werden wie oben beschrieben Daten unifiziert. Damit wird formal beschreibbar, welche Daten konsistent und welche inkonsistent sind. Weiterhin ergibt sich für konsistente Daten eine eindeutige Repräsentation für deren gemeinsamen Informationsgehalt. Nicht betrachtet wird der Fall, in dem für inkonsistente Daten spezielle Maßnahmen getroffen werden. Die Wiederherstellung der Konsistenz ist nicht einfach und auch nicht immer möglich. Die *fehlertolerante* Zusammenführung von Information ist ein komplexer Themenbereich für sich, der in dieser Arbeit nicht behandelt wird.

Durch verschiedene Untersuchungen wurde gezeigt, daß auch die Objektorientierung mit dem Problem der Datenkonsistenz in verteilten Systemen zu kämpfen hat ([Boger et al. 1999b]). Entfernte Methodenaufrufe und Migrationsmechanismen für Objekte können zu verteilten Objektkopien führen, deren Konsistenz nicht mehr zugesichert werden kann. Es muß durch entsprechende Konzepte dafür gesorgt werden, daß Objekte bei der Migration ihre Identität behalten ([Boger et al. 1999a]). Um Konsistenzprobleme zu verhindern, kann verlangt werden, daß ein Objekt permanent an einem Ort vorhanden ist. Da dies ähnliche Nachteile wie die pessimistische Konsistenzkontrolle nach sich zieht, wird in neueren Ansätzen ermöglicht, den (eindeutigen) Ort eines Objekts zur Laufzeit zu ändern (*Objektmigration*, siehe z.B. [Boger 1999]). Durch die in dieser Arbeit vorgestellten Grundlagen kann eine weitere Möglichkeit in Betracht gezogen werden. Analog zu der oben diskutierten optimistischen Konsistenzkontrolle könnten Objektkopien erlaubt werden, die durch Unifikation oder einen ähnlichen Mechanismus wieder zusammengeführt werden können.

Die Konsistenz verteilter Information wurde in diesem Abschnitt als ein wichtiges Phänomen in verteilten Systemen anhand eines Beispiels motiviert.

1.1.2 Nebenläufigkeit und Synchronisation verteilter Prozesse

In verteilten Systemen werden Anwendungen auf viele verschiedene Prozessoren verteilt. Damit ergibt sich (ohne weitere Vorkehrungen zur Synchronisation) eine

inhärente Nebenläufigkeit verteilter Systeme.

Das durch Nebenläufigkeit implizierte Konzept des verzweigenden Steuerflusses[3] (*multi-threading*) wird in Modellierungs- wie Programmiersprachen heutzutage nur unzureichend unterstützt ([Douglass 1999], [Boger 1999]). Den Normalfall stellt immer noch die Annahme dar, daß eine Anwendung sequenziell abläuft.

Vor allem die weit verbreiteten *imperativen* Sprachen wie Pascal, Basic, Fortran, C, C++, Java (auch Algol-Sprachen genannt, [Louden 1994]) und viele andere eignen sich zur nebenläufigen Abarbeitung nur bedingt, da sie auf dem Prinzip des *synchronen* Prozedur- bzw. Methodenaufrufs beruhen. Dieses Prinzip wurde auf Modellierungssprachen übertragen. In UML gibt es viele abkürzende Notationen für synchrone Aufrufe, doch die Semantik der zur Darstellung asynchroner Aufrufe hinzugenommenen Konstrukte ist unbefriedigend (vergleiche [Engels et al. 2000]).

Wie im vorherigen Abschnitt für die informationsorientierte Modellierung verteilter Systeme soll auch für die prozeßorientierte Modellierung anhand eines ausgewählten Gebiets motiviert werden, daß sich verteilte Systeme zum Teil grundsätzlich von klassischen Rechensystemen unterscheiden.

In der Programmierung wie in der Modellierung wurde versucht, die bekannten Techniken durch punktuelle Erweiterungen auf verteilte Systeme zu übertragen. Dies stellt insofern einen Fortschritt dar, als die Programmierung verteilter Systeme scheinbar ohne Paradigmenwechsel möglich wird. Dennoch muß der grundsätzlich andere Charakter verteilter Systeme beachtet werden, was zu differenzierteren Herangehensweisen geführt hat. Ein Beispiel für ein solches Vorgehen ist der entfernte Prozeduraufruf (*remote procedure call*, RPC, ([B.J.Nelson 1981, Schill 1992])), in der Objektorientierung auch entfernter Methodenaufruf (*remote method invocation*, RMI) genannt. Beide Arten sollen hier unter dem Begriff *entfernter Aufruf* zusammengefaßt werden. Der entfernte Aufruf wird im folgenden auf konzeptueller Ebene näher betrachtet.

Während eines synchronen Aufrufs blockiert der Aufrufer, bis der Aufgerufene ein Ergebnis geliefert hat, und sei es auch nur das Ergebnis, daß eine Beendigung der Aufgabe erfolgt ist. In verteilten Systemen hat ein solches Vorgehen mehrere Nachteile:

- Die End-Synchronisation ist nicht trivial. Antwortnachrichten können ausbleiben oder verlorengehen. In einem solchen Fall ist es unerwünscht, daß der Aufrufer weiterhin blockiert.
- Die Komplexität des entfernten Aufrufs soll durch die Darstellung als synchroner Aufruf vor dem Modellierer bzw. Anwendungsprogrammierer verborgen werden. Durch im verteilten Fall hinzugekommene Fehlermöglichkeiten (wie

[3] Unter anderem in [Hitz und Kappel 1999] und [Jablonski et al. 1997] wird von „Kontrollfluß“ gesprochen. Der Autor hält dies aber für eine unangemessen direkte Übersetzung des englischen Begriffs *control*. Das deutsche Wort *Steuerung* trifft die Intention von *control* besser, da der Ablauf von Prozesses nicht kontrolliert (also überprüft), sondern (vom Benutzer oder dem System) gesteuert wird.

nicht überall im System verfügbare Daten, unterschiedliche Ausführungsumgebungen und weitere Phänomene) wird eine vollkommene Transparenz der Verteilung inzwischen als nicht wünschenswert angesehen ([Boger 1999]).

- Synchrone Aufrufe stellen nur einen von vielen Fällen der Kommunikation in verteilten Systemen dar, der aufgrund der inhärenten Nebenläufigkeit zudem der unnatürlichste ist ([Lamersdorf 1994], [Boger 1999]).

- In Einprozessor-Systemen wirkt sich ein blockierender Aufruf im allgemeinen nicht nachteilig aus, da die Prozessorleistung zur Ausführung der aufgerufenen Prozedur bzw. Methode genutzt wird. Während des Wartens auf die Beendigung eines entfernten Aufrufs kann sich der Prozessor hingegen im Leerlauf befinden, falls nicht durch *multi-threading* andere lokale Prozesse Rechenzeit zugeteilt bekommen. Es kann somit vorkommen, daß Ressourcen verschwendet werden.

Die transparente Realisierung entfernter Aufrufe scheint zunächst für Modellierer und Programmierer einfacher, was sich aber aufgrund der genannten Nachteile relativiert. Ein verteiltes System *ist* durch Nebenläufigkeit und Synchronisation (und viele weitere der oben genannten Eigenschaften) komplexer als ein monolithisches, sequenzielles System. Es gilt, diese Komplexität zu akzeptieren und entsprechende Konzepte und Techniken zu ihrer Programmierung *und* Modellierung zur Verfügung zu stellen.

In Anlehnung an [Engels et al. 2000] werden als Beispiel für eine angemessenere Modellierung der entfernten Kommunikation in verteilten Systemen im folgenden verschiedene Möglichkeiten anhand einfacher Petrinetzmodelle skizziert. Dies zeigt zum einen die Vielfalt der Kommunikationsmöglichkeiten in verteilten Systemen, zum anderen die Eignung von Petrinetzen zur deren exakter Modellierung.

Nach [Engels et al. 2000] werden Start und Ende einer Interaktion getrennt als sogenannte *Interaktionsmuster* (*interaction patterns*) modelliert, da sich unterschiedliche Kommunikationsarten für Start und Ende orthogonal kombinieren lassen.

Abbildung 1.6 zeigt drei[4] Muster für den Start einer Interaktion nach [Engels et al. 2000], S. 311. Die dort verwendete informale Notation für Interaktionsmuster wurde hier auf Petrinetze übertragen. Die drei Interaktionsmuster in Abbildung 1.6 stellen folgende Fälle dar:

(a) Der Start einer Interaktion wird *synchron* (*synchronous*) genannt, wenn der Sender sofort nach dem Absenden der Nachricht blockiert, bis der Empfänger die Nachricht akzeptiert hat.

[4] In [Engels et al. 2000] wird sowohl für den Start als auch das Ende einer Interaktion als vierte Möglichkeit *eingeschränkt asynchron* (*restricted asynchronous*) betrachtet. Als Beispiele für unterschiedliche Interaktionsmuster sollen die anderen drei Möglichkeiten hier ausreichen.

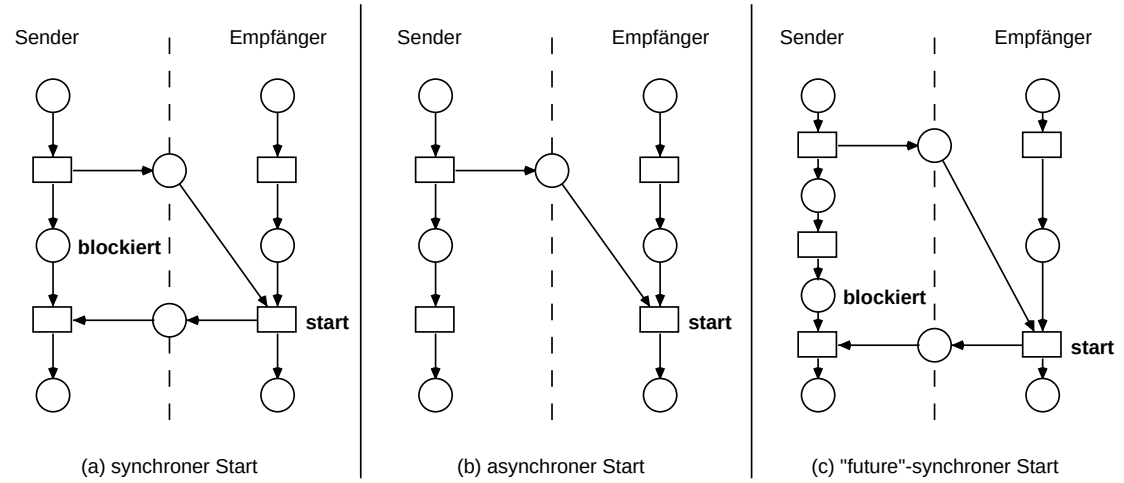

Abbildung 1.6: Drei Muster für den Start einer Interaktion nach [Engels et al. 2000], S. 311, hier als Petrinetze dargestellt.

(b) Der Start einer Interaktion wird *asynchron* (*asynchronous*) genannt, wenn der Sender nach dem Absenden der Nachricht mit anderen Aktivitäten fortfährt, unabhängig davon, wann und ob der Empfänger die Nachricht akzeptiert.

(c) Der Start einer Interaktion wird *zukünftig-asynchron* (*future asynchronous*) genannt, wenn es dem Sender erlaubt ist, nach dem Absenden der Nachricht mit anderen Aktivitäten fortzufahren, aber in einem späteren Zustand blockiert, bis der Empfänger die Nachricht akzeptiert hat.

Analog dazu können für das Ende einer Interaktion entsprechende Fälle unterschieden werden (siehe [Engels et al. 2000], S. 312). Durch Kombination der Interaktionsmuster für Start und Ende einer Interaktion ergeben sich *vollständige Interaktionsmuster*.

Abbildung 1.7 zeigt drei Beispiele für vollständige Interaktionsmuster nach [Engels et al. 2000], S. 313, wobei hier wiederum Petrinetze zur Darstellung benutzt werden. Das erste Beispiel in (a) modelliert den oben beschriebenen Fall des synchronen Prozeduraufrufs. Dieser stellt damit einen Sonderfall der allgemeinen Betrachtung von Interaktionsmustern dar. Das zweite Beispiel in Abbildung 1.7 (b) zeigt einen vollkommen asynchronen Aufruf, der auch als *one-way* Aufruf bezeichnet wird ([Boger 1999]). Der dritte Fall in Abbildung 1.7 (c) zeigt, daß auch bei asynchron startender Interaktion eine spätere Synchronisation möglich ist und wird auch als *Future*-Nachricht bezeichnet ([Engels et al. 2000]).

Durch die Betrachtung verschiedener Arten von Kommunikation und Interaktion in verteilten Systemen wurde in diesem Abschnitt motiviert, daß Paradigmen der

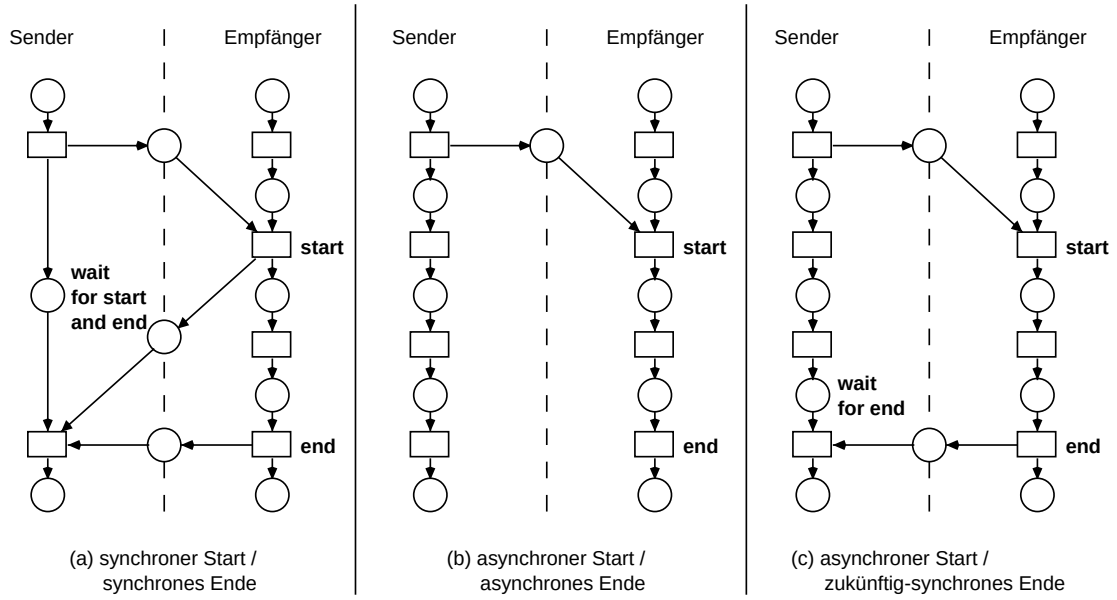

Abbildung 1.7: Drei Beispiele für vollständige Interaktionsmuster nach [Engels et al. 2000], S. 313, hier als Petrinetze dargestellt.

klassischen Modellierung und Programmierung nicht immer auf den verteilten Fall übertragen werden können.

Während die im vorherigen Abschnitt diskutierten Problemstellungen eine spezielle informationsorientierte Modellierung verteilter Systeme erfordern, stellen Nebenläufigkeit und Synchronisation Phänomene dar, die eine prozeßorientierte Modellierung erfordern. Die in dieser Arbeit vorgestellten FSNets sind eine spezielle Variante höherer Petrinetze und eignen sich damit zur Prozeßmodellierung (siehe Abschnitt 3.2). Es wurde bei der Definition von FSNets darauf geachtet, daß diese in ihrer Grundstruktur mit den klassischen Petrinetzen übereinstimmen. Dadurch bleibt die Eigenschaft von Petrinetzen erhalten, daß diese sich zur Modellierung von Nebenläufigkeit und Synchronisation eignen.

1.2 Modellierung verteilter Systeme: Information versus Prozesse

Die Modellierung von Anwendungssystemen stellt in der Informatik eine zentrale Fragestellung dar. Je mehr Erfahrungen in einem Gebiet der Informatik zur Verfügung stehen, desto besser wird die Modellierung unterstützt. So kann beispielsweise die Datenbankmodellierung auf ein breites, standardisiertes Repertoire

von Modellierungstechniken zurückgreifen (siehe z.B. [Date 1995]). In dieser Arbeit stehen verteilte Systeme als ein Gebiet der Informatik im Mittelpunkt, das neue Herausforderungen an die Analyse und den Entwurf von Anwendungssystemen stellt. Das World Wide Web hat sich in den letzten Jahren von einem verteilten, nur lose zusammenhängendem Hypertext-Informationssystem zu einer Plattform für das *network computing* ([Sun 2000a]) entwickelt, auf der verteilte Anwendungen auf Millionen von Rechnern laufen. Für solche Anwendungen stehen heute zur Bewältigung der Probleme der Verteilung bereits viele Technologien aus den Bereichen der Programmiersprachen, Systemumgebungen (*middleware*) und der Softwarearchitektur zur Verfügung. Es gibt aber noch keine angemessenen, allgemein einsetzbaren *Spezifikationsansätze* für verteilte Anwendungen.

Ein Ansatz zur Systemspezifikation umfaßt nach [Moldt 1996], S. 29 ff., mehrere *Facetten*, die dort als *Methoden*, *Techniken*, *Werkzeuge*, *Anwendungen* und *Ressourcen* bezeichnet werden und in einem *Kontext* angesiedelt sind. Die in der vorliegenden Arbeit in bezug auf die Spezifikation verteilter Systeme behandelten Facetten sind Techniken, Werkzeuge und Anwendungen. Ressourcen im Sinne eines Ansatzes nach Moldt werden teilweise für den konkreten Fall der Geschäftsprozeßmodellierung betrachtet (siehe Abschnitt 5.1). Methoden, die an anderer Stelle (siehe z.B. [Jacobson et al. 1995]) auch *Vorgehensmodelle* genannt werden, sind nicht Thema dieser Arbeit. Es werden im folgenden einige Verweise auf Arbeiten in diesem Gebiet gegeben. Bei der Erstellung der Anwendungsbeispiele in Kapitel 5 wird systematisch vorgegangen, dies erhebt aber keinen Anspruch auf Allgemeinheit und stellt damit keine Methodik zur Verfügung.

Erste Vorgehensmodelle für bestimmte Bereiche der verteilten Systeme sind bereits vorgeschlagen worden. Beispielsweise wird in [Conallen 2000] ein Vorgehensmodell für Web-Anwendungen gegeben, das auf dem *Rational Unified Process* (RUP, [Jacobson et al. 1999]) basiert, der UML als Modellierungstechnik verwendet. Dies zeigt die Bedeutung von UML als Modellierungstechnik im allgemeinen, wobei wir in Kapitel 2 sehen werden, daß UML sich für spezielle Problemstellungen bei der Modellierung von Verteilung nur bedingt eignet. Als weiteres Vorgehensmodell sei auf Catalysis ([D'Souza und Wills 1998]) hingewiesen, welches ebenfalls auf UML basiert. Catalysis ist zwar nicht auf verteilte Systeme spezialisiert, legt aber viel Wert auf die Modellierung von Objektgraphen, die hier als eine wesentliche Basis für die Modellierung verteilter Kopien angesehen wird (siehe Kapitel 2). Weiterhin existieren bereits erste Vorgehensmodelle für die agentenorientierte Spezifikation ([Kinny et al. 1996], [Brenner et al. 1998]), die ebenfalls für verteilte Anwendungen interessant sind.

Es ist ein grundlegendes Prinzip der Modellierung, daß man ein System mit mehreren Teilmodellen aus verschiedenen *Sichten* (nicht zu verwechseln mit der oben erwähnten *Objektsicht*) beschreibt (siehe dazu [Moldt 1996]). Ein Teilmodell sollte sich dabei stets auf eine Sicht konzentrieren. Es gibt unterschiedliche Auffassungen, welche Sichten für Anwendungssysteme sinnvoll sind; eine klassische Unterteil-

lung ist die zwischen *statischer* und *dynamischer Sicht* ([Rumbaugh et al. 1991, Coad und Yourdon 1991]). Dabei beschreibt die statische Sicht das System zu einem beliebigen Zeitpunkt oder zeitlos, also unabhängig von vorangegangenen oder nachfolgenden Zuständen, während die dynamische Sicht Beziehungen und Übergänge zwischen Systemzuständen zu verschiedenen Zeiten modelliert. Oft wird weiterhin noch eine *funktionale Sicht* getrennt betrachtet (siehe z.B. [Moldt 1996]), die Funktionen und Dienste eines Systems ohne deren dynamisches Verhalten betrachtet. In objektorientierten Vorgehensweisen gibt es orthogonal zu dieser Unterteilung die *objekt-* und die *prozeßorientierte Sicht* ([Hitz und Kappel 1999, OMG 2000]). Dabei beschreibt die objektorientierte Sicht die (statische) Struktur von Objekten zusammen mit der Dynamik innerhalb einzelner Objekte (*intra*-Objektdynamik), die prozeßorientierte Sicht dagegen (dynamische) Abläufe, die nicht einem einzelnen Objekt zugeordnet werden können (*inter*-Objektdynamik) und gegebenenfalls statische Aspekte wie Rollen, die nur in diesen Prozessen eine Bedeutung haben. Diese Trennung gibt es bei der objektorientierten *Programmierung* nicht, in der letztendlich das gesamte dynamische Verhalten bestimmten Objekten bzw. Klassen zugeordnet werden muß.

An Stelle von statischer Modellierung spricht man auch von *Daten-* oder *Informationsmodellierung*. Dabei schließt die statische Sicht die Modellierung von Schnittstellen mit ein, die im dynamischen Verhalten des System wieder auftreten. Die Schnittstellen gehören nicht zum dynamischen Teil des Modells, da es sich in der klassischen Objektorientierung lediglich um statische Schnittstellen handelt[5].

Diesem grundlegenden Schema entsprechend wird die Modellierung verteilter Systeme in dieser Arbeit in eine informations- und eine prozeßorientierte Sicht unterteilt.

Der Begriff „informationsorientiert“ wird bevorzugt, da in modernen Ansätzen und Systemen von verschiedenste Arten von Information ausgegangen wird, denen der Begriff „Daten“ nicht mehr gerecht wird. Beispielsweise spricht man statt von „Datenbanken“ inzwischen oft von „Wissensmanagement“ ([Date 1995]) und statt Nachrichten, die zwischen Objekten versandt werden, von Informationsaustausch oder gar Sprechakten zwischen Agenten ([Shoham 1997], vergleiche Abschnitt 5.3.1).

Die „dynamische“ Sicht wird hier als „prozeßorientiert“ bezeichnet, da Prozesse in der Petrinetz-Terminologie die Dynamik *verteilter* Systeme beschreiben. In der klassischen dynamischen Sicht, wie sie beispielsweise von Zustandsautomaten repräsentiert wird (siehe Abschnitt 3.1.2), wird die Dynamik eines Systems beschrieben durch *Zustände* und *Ereignisse*, die *Zustandsübergänge* auslösen. Petrinetze stellen eine Verallgemeinerung dieses Ansatzes dar, da sie zusätzlich die *Nebenläufigkeit* von Ereignissen spezifizieren können (siehe u.a. [Jessen und Valk 1987],[Baumgarten 1990],[Aalst 1998]).

[5]Schnittstellen, welche die Dynamik eines Objekts beschreiben (sogenannte Verhaltensschnittstellen, *behavioural interfaces*), werden beispielsweise in [Aalst und Basten 1997, Aalst und Anyanwu 1999] betrachtet.

Modellierungssicht	modellierte Aspekte	Technik
informationsorientiert	Information, Typen, Daten, Rollen, (Daten-)Objekte, Schnittstelle, Wissen, Bedingungen, Regeln	Feature Structures
prozeßorientiert	Prozesse, Steuerung, Abläufe, Vorgänge, Verhalten, Protokolle	Petrinetze

Tabelle 1.1: Informations- und prozeßorientierte Modellierungssichten und die in dieser Arbeit verwendeten Techniken.

Die Unterteilung in diese beiden Sichten ist nicht die einzig mögliche oder alles umfassende Lösung. Es handelt sich aber mit Sicherheit um zwei Sichten, denen bei der Modellierung verteilter Systeme eine zentrale Bedeutung zukommt. Wie im vorherigen Abschnitt diskutiert und motiviert wurde, weisen beide Sichten in verteilten Systemen gewisse Besonderheiten auf, die im folgenden aus der Modellierungsperspektive betrachtet werden.

Tabelle 1.1 zeigt in einer Gegenüberstellung, welche Aspekte einer Domäne informations- bzw. prozeßorientiert modelliert werden. Die folgenden Abschnitte gehen auf beide Sichten nacheinander ein.

Informationsorientierte Modellierung

Die informationsorientierte Sicht entspricht im wesentlichen der oben genannten statischen Sicht und widmet sich dementsprechend der Modellierung von Information, Typen, Daten, Objekten, Rollen und Schnittstellen. Dabei ist zu beachten, daß das *Verhalten* von Objekten nicht Teil der informationsorientierten Modellierung ist, weshalb in dieser Arbeit auch von *Daten*objekten gesprochen wird, um zu betonen, daß Objekte ohne ihr Verhalten gemeint sind.

Als ein wichtiger Unterschied zum üblichen Gebrauch des Begriffs der statischen Sicht schließt die informationsorientierte Sicht Aspekte wie Wissen, Bedingungen und Regeln mit ein.Damit werden Querverbindungen zur *Künstlichen Intelligenz* (KI), vor allem zum Gebiet der *Wissensrepräsentation* deutlich. *Regeln* befinden sich am äußersten Ende der Informationsmodellierung, da sie durch entsprechende Interpretation zu einem bestimmten Verhalten führen können. Eine Regel an sich wird hier aber noch als Information aufgefaßt.

Oben wurde als spezielles Problem verteilter Systeme die Konsistenzhaltung von Daten oder Information beleuchtet. Dieses Problem wird (noch stärker als bei der prozeßorientierten Modellierung in bezug auf entfernte Aufrufe) in verfügbaren Modellierungstechniken „wegabstrahiert“, statt es beschreibbar zu machen. Einige wenige Möglichkeiten der Darstellung von verteilten Objekten gibt es in UML (siehe Abschnitt 2.2.3). Durch das *Verteilungsdiagramm* können gewisse *statische* Aspekte der Verteilung von Information modelliert werden. Mithilfe der *Object Constraint Language* (OCL), die einen Teil von UML darstellt, ist die Beschreibung von Einschränkungen auf Objektgraphen sowie Vor- und Nachbedingungen von Operationen möglich. Damit können Konsistenzeigenschaften von Daten spezifiziert werden, es ist aber weder eine Beschreibung der Modifikation von Daten möglich, noch ist OCL in anderer Weise auf verteilte Systeme spezialisiert.

In der hier vorgeschlagenen Modellierungstechnik der FSNets werden Typhierarchien (*order sorted types*) und Feature Structures zur Informationsmodellierung eingesetzt. Feature Structures können wie im vorherigen Abschnitt erwähnt als Objektsichten verstanden werden. Im Zusammenhang mit der Konsistenzsicherung ist aber auch eine Verwandtschaft mit OCL ([OMG 2000], [Warmer und Kleppe 1998]) zu erkennen. Eine Feature Structure stellt wie ein OCL-Ausdruck eine logische Beschreibung dar, welche durch Angabe bestimmter Eigenschaften die Struktur eines Objekts bzw. Objektgraphen einschränkt (*constraint*). Der Vorteil von Feature Structures gegenüber OCL ist, daß durch Feature Structures Information *normalisiert* vorliegt, so daß sie beispielsweise verglichen werden kann. Eine Aussage darüber, ob zwei gegebene OCL-Ausdrücke äquivalent sind, ist aufgrund deren höherer Komplexität nicht

möglich. Es kann dagegen eindeutig festgestellt werden, ob eine Feature Structure spezieller als eine andere ist, dieselbe Information repräsentiert oder ob der Informationsgehalt der Feature Structures unvergleichbar ist. Dafür bietet OCL eine höhere Ausdrucksmächtigkeit, inbesondere in bezug auf mehrwertige Attribute.

Feature Structures haben ihre Wurzeln in der KI und wurden ursprünglich als Wissenrepräsentationstechnik eingeführt ([Smolka 1988, Pollard und Sag 1987, Carpenter 1992]) und zur Logikprogrammierung genutzt ([Aït-Kaci et al. 1992]). Demnach sind sie insbesondere für die oben betrachtete Repräsentation von Bedingungen und Regeln geeignet. Feature Structures werden in der KI vor allem im Bereich der Modellierung natürlicher Sprache eingesetzt ([Shieber 1986, Matthiessen und Kasper 1987, Vijay-Shanker und Joshi 1988, Pollard und Sag 1994]), die Voraussetzung für Sprachverstehen und -generierung ist. Feature Structures werden aber auch für die Logikprogrammierung ([Aït-Kaci und Nasr 1986]) und Problemstellungen aus der Softwaretechnik wie Konfigurationsmanagement ([Zeller 1994]) und Versionsverwaltung ([Zeller und Snelting 1996], [Zeller und Snelting 1997]) eingesetzt. Dies ist ein weiteres Beispiel für den auch im Bereich der Agenten zu erkennenden Trend, KI-Techniken in der Softwaretechnik einzusetzen ([Brenner et al. 1998], [Müller 1993], siehe auch 5.3.1). Diese Arbeit zeigt als neuen Beitrag, daß Feature Structures als formale Grundlage für Objektgraphen herangezogen werden können.

Prozeßorientierte Modellierung

In Tabelle 1.1 werden verschiedene Aspekte der Domäne aufgezählt, welche in einer prozeßorientierten Modellierung betrachtet werden. Die dynamische Sicht klassischer Systeme betont den *Steuerfluß* eines Ablaufs oder Vorgangs. Dazu gehört der Fokus auf die sequenzielle Reihenfolge, auf Entscheidungen (*choice*) und Aufrufe von Unterroutinen. In verteilten Systemen treten in nicht-trivialen Fällen wie in Abschnitt 1.1.2 dargestellt zusätzlich Nebenläufigkeit und Synchronisation auf. Die Dynamik verteilter Systeme kann deshalb am besten durch prozeßorientierte Techniken beschrieben werden, mit denen alle diese Konzepte dargestellt werden können.

Fortgeschrittenere prozeßorientierte Konzepte betrachten einen Ablauf als *Verhalten* eines Objekts oder wiederum selbst als Objekt. Abläufe, die als eigenständige Objekte ausgelagert werden, haben den Charakter eines *Protokolls* ([Ciancarini 1999, Laue et al. 2000, Tu et al. 2000c]). Ein Protokoll in diesem Sinne steuert das Verhalten eines anderen Objekts, im Gegensatz zum Strategy-Entwurfsmuster ([Gamma et al. 1995]), welches das Objekt selbst steuert. Dies erhöht die Flexibilität des Objektverhaltens, da sich das Protokoll-Objekt zur Laufzeit austauschen läßt.

Auch für die Modellierung von Systemdynamik spielt UML in der Praxis eine entscheidende Rolle ([Engels et al. 2000]). Mit Interaktions-, Zustands- und Aktivitäts-

diagrammen stellt UML ein Repertoire zur dynamischen Modellierung zur Verfügung (siehe Abschnitt 3.1.1). Ähnlich den UML-Zustandsdiagrammen gibt es verschiedene weitere, auf endlichen Automaten basierende Techniken wie StateCharts ([Harel 1987]) und SOCCA ([Engels et al. 2000]), die zur zustands- und ereignisorientierten Modellierung eingesetzt werden. Prozeßorientierte Modellierungstechniken stammen hauptsächlich aus dem Bereich der *Geschäftsprozeßmodellierung*. Hier sind Techniken wie EPK und Aktivitätenmodelle zu nennen (siehe Abschnitt 3.1).

Die meisten dieser in der Praxis verwendeten Ansätze zeigen den Charakter von Modellierungssprachen, die zur Kommunikation und zur Veranschaulichung von Modellen geeignet sind. Vor allem im Bereich der Dynamik müssen sie sich aber eine mangelnde Formalisierung vorwerfen lassen. Wenn man bedenkt, daß UML einen Standard darstellt, sind die bekannten syntaktischen und semantischen Schwächen erstaunlich (siehe dazu [Precise UML Group 2000], insbesondere [Harel und Rumpe 2000], [Maibaum und Rumpe 2000] und verschiedene Beiträge in [France und Rumpe 1999] und [Evans et al. 2000]). Eine zu genaue Syntax und Semantik wird zwar vor allem in der Analyse oft als zu einschränkend empfunden. Ohne eine formale oder zumindest implementierte *operationale Semantik* ist aber eine *Validation* oder *Ausführung von Modellen* nicht möglich. Die Modellierungssprache wird dann zu einer diagrammatischen Veranschaulichung, die aufgrund fehlender Semantik nicht eindeutig interpretierbar sind, weder für einen Betrachter noch für ein Software-Werkzeug. Es kann damit zu Ungenauigkeiten und Mißverständnissen kommen, und es ist kein direkter Bezug zu einer späteren Realisierung als Anwendungssystem gewährleistet. Auch läßt sich wenig über die Korrektheit und Vollständigkeit des Modells aussagen ([Precise UML Group 2000], [Kennedy 1999]).

In dieser Arbeit wird eine exakte, formal definierte operationale Semantik der Modellierungstechnik gefordert. Petrinetze sind für ihre operationale Semantik und ihre Stärken bei der prozeßorientierten Modellierung nebenläufiger und verteilter Systeme bekannt ([Jessen und Valk 1987],[Baumgarten 1990],[Jensen 1997]), was an dem in diesem Abschnitt gegebenen Beispiel nochmals demonstriert wurde.

Die Unifikation von Information, die im vorherigen Abschnitt am Beispiel einer optimistische Konsistenzkontrolle eingesetzt wurde, findet in verteilten Systemen *innerhalb von Prozessen* statt. In dieser Arbeit werden theoretische Grundlagen aus dem Bereich der Feature Structures und der Petrinetze in Verbindung gebracht, um dieses Phänomen formal beschreiben zu können. Der Unterschied zwischen referenzierten und kopierten Daten oder Objekten wird durch die Einführung von Referenz- und Wertsemantiken formalisiert. Auf dieser Basis entsteht eine Modellierungstechnik, in der Systeme mit verteilten Objektkopien modelliert und, da die Modelle durch entsprechende Werkzeugunterstützung ausführbar sind, realisiert werden können. Damit wird ein Beitrag zum Verständnis und zur Modellierung von verteilten Systemen geleistet.

1.3 Aufbau der Arbeit

Diese Arbeit stellt verschiedene Modellierungstechniken in bezug auf verteilte Systeme vor, schlägt eine neu entwickelte Modellierungstechnik vor, die formal definiert und mit Hilfe eines prototypischen Werkzeug zur Verfügung gestellt wird und zeigt den praktischen Einsatz an Beispielen aus dem Bereich der verteilten Systeme.

Die verschiedenen Modellierungstechniken werden nach informations- und prozeßorienterten Aspekten unterteilt vorgestellt.

Die informationsorientierte Modellierung wird in Kapitel 2 betrachtet. Klassen und Typen werden als Grundlage in Abschnitt 2.1 eingeführt und formalisiert. Für die formal definierten Konzeptsysteme wird eine UML-Notation vorgestellt. In Abschnitt 2.2 werden Objekte, welche auf einer Klassen- oder Typhierarchie basieren, am Beispiel von UML eingeführt. Dabei wird insbesondere auf Objektsichten und Darstellung der Verteilung von Objekten eingegangen. Abschnitt 2.3 stellt den hier zur Informationsmodellierung verwendeten Ansatz der Feature Structures vor, der eine zentrale Rolle in der entwickelten Modellierungstechnik spielt. Feature Structures werden formal definiert und zu den objektorientierten Konzepten und Notationen aus den vorhergehenden Abschnitten in Bezug gesetzt. Der letzte Abschnitt 2.4 stellt die Implementierung einer Feature-Structure-Bibliothek vor, die im Rahmen dieser Arbeit als Voraussetzung zur Realisierung eines Modellierungswerkzeugs für FSNets erstellt wurde.

Kapitel 3 stellt Konzepte und Techniken der prozeßorientierten Modellierung vor. Nach einer Einführung in die entsprechenden Konzepte und Techniken von UML und weiterer ausgewählter Techniken wie Aktivitätenmodelle und Ereignisgesteuerter Prozeßketten in Abschnitt 3.1 liegt der Schwerpunkt des Kapitels auf Petrinetzen. Abschnitt 3.2 stellt klassische Petrinetze wie B/E-Netze, S/T-Netze und gefärbte Petrinetze sowie erweiterte Kantenarten vor. Abschnitt 3.3 gibt eine Einführung in das neuere Paradigma der „Netze in Netzen“, das verschiedenen höheren Petrinetzen zugrundeliegt. In Abschnitt 3.4 wird die Geschäftsprozeßmodellierung vor allem unter dem Blickwinkel eines Anwendungsgebiets für Petrinetze betrachtet. Abschnitt 3.5 stellt das erste Petrinetzwerkzeug vor, das auf dem Paradigma der Netze in Netzen aufbaut und vom Autoren mitentwickelt und als Ausgangsbasis für ein FSNet-Werkzeug herangezogen wurde.

In Kapitel 4 wird die neue Modellierungstechnik der Feature-Structure-Netze, kurz FSNets, Schritt für Schritt entwickelt. Dabei wird ein den gefärbten Petrinetzen ähnlicher Basisformalismus als Basis-FSNets in Abschnitt 4.1 unter Berücksichtigung einer Schaltfolgensemantik definiert. Eine eingeschränkte Variante dieses Formalismus, das elementare FSNet (EFSNet), wird in Abschnitt 4.2 definiert, wobei die Definition der Schaltregel und die Darstellung von Prozessen hier vollständig durch Feature Structures erfolgt. Anhand von EFSNets werden Fragestellungen wie die Unterscheidung von Wert- und Referenzsemantik, die Definition eines universellen elementaren FSNets und einer Prozeßsemantik behandelt. FSNets zeigen durch

die Verwendung von Graphunifikation Parallelen zum Ansatz der Netze in Netzen. Abschnitt 4.3 zeigt zwei Varianten, wie Netze in Netzen mit mit FSNets dargestellt werden können. Abschnitt 4.4 geht auf das erweiterte Modell der höheren FSNets ein, die durch eine Kombination mit Konzepten der Referenznetze den praktischen Einsatz von FSNets verbessern. Das Kapitel schließt mit der Vorstellung des implementierten Modellierungswerkzeugs, mit dem höhere FSNets erstellt und ausgeführt werden können.

Kapitel 5 zeigt die Anwendung der in Kapitel 4 entwickelten Modellierungstechnik anhand praxisrelevanter Beispiele aus dem Bereich der verteilten Systeme. Als repräsentative Bereiche werden FSNets zur Modellierung, Simulation und Ausführung von Geschäftsprozessen (Abschnitt 5.1), elektronischen Verträgen (Abschnitt 5.2) und intelligenten Software-Agenten (Abschnitt 5.3) eingesetzt.

Kapitel 6 schließt mit einer Zusammenfassung (Abschnitt 6.1, die den Schwerpunkt auf die erzielten Ergebnisse legt, und einem Ausblick auf weitere Forschung (Abschnitt 6.2.

Kapitel 2

Informationsorientierte Modellierung

In diesem Kapitel werden Aspekte informationsorientierter Modellierungskonzepte und -techniken vertieft. Die Diskussion basiert auf der heutzutage verbreitetsten Art, Systemmodelle zu erstellen: der objektorientierten Analyse. Auch wenn objektorientierte Modellierung keine reine Informationsmodellierung darstellt, da auch Operationen betrachtet werden, kann man bei der reinen Informationsmodellierung von einem objektorientierten Vorgehen sprechen, wenn man Vererbung, Attribute und Assoziationen betrachtet.

Durch UML ([OMG 2000]) gibt es einen Notationsstandard, der durch seinen Einfluß nicht nur die Syntax, sondern auch die Semantik objektorientierter Modelle prägt. Allerdings wurde die Spezifikation von UML als noch zu unpräzise kritisiert (siehe z.B. [Precise UML Group 2000], [Harel und Rumpe 2000], [Maibaum und Rumpe 2000] oder [Kennedy 1999]). Dennoch wird in dieser Arbeit UML als konsolidierter Stand der Technik im Gebiet der objektorientierten Modellierungstechniken akzeptiert und dementsprechend als Ausgangspunkt der Betrachtung gewählt.

Betrachtet man lediglich die informationsorientierten Aspekte der Modellierung, so lassen sich diese in zwei weitere Bereich unterteilen: Einerseits werden durch Typen und Klassen Kategorien von Entitäten der Anwendungsdomäne modelliert. Andererseits können auch die Entitäten selbst oder Beispiele für Entitäten modelliert werden. Dabei hängt die Reihenfolge dieser Modellierungsaspekte vom Vorgehensmodell ab.

Klassen und Typen werden in Abschnitt 2.1 behandelt. Hier liegt vor allem in der Vererbung die Erweiterung von objektorientierten gegenüber beispielsweise relationalen Modellierungstechniken (*entity-relationship-model*, ER-Modell, siehe z.B. [Chen 1976]). Gemeinsam basieren sie jedoch auf der Annahme, daß Informationseinheiten durch Eigenschaften (*properties, features*), auch Attribute (*attributes*) genannt, und Assoziationen (*associations*) zwischen diesen Informationseinheiten be-

schrieben werden. Die Ansätze unterscheiden sich allerdings in bezug auf die Struktur der Werte von Eigenschaften und den Zusammenhang zwischen Typen und Eigenschaften. Diese Art der Informationsmodellierung wird hier durch hierarchische Typsysteme (*order sorted types*, [Schmidt-Schauss 1989]) mit Attributen formalisiert, wobei wir auf einen Ansatz von Carpenter ([Carpenter 1992]) zurückgreifen. Dieser Ansatz wird mit den Klassendiagrammen von UML in Bezug gesetzt. Dabei werden insbesondere Eigenschaften wie Subtypbeziehungen und Assoziationen zwischen Typen betrachtet. Wird Information ausschließlich über Typen modelliert, kann bereits auf dieser Ebene Typunifikation (*join*) eingesetzt werden, um Information zu vereinigen.

Neben Klassen Typen werden wie oben sagt auch die eigentlichen Datenobjekte oder Entitäten modelliert. Auch für die Modellierung einzelner Objekte oder abstrakte *Objektsichten* stellt UML spezielle Notationen zur Verfügung. Weiterhin wird auf die Darstellung von *Verteilung* in UML eingegangen, die praktisch nur auf der statischen Ebene möglich ist.

Als spezieller Ansatz dieser Arbeit werden Feature Structures in Abschnitt 2.3 als mächtige und dennoch verständliche und handhabbare Technik zur Informationsmodellierung eingeführt. Feature Structures passen insofern nicht in das oben angegebene Schema der Typen und Objekte, da sie, ähnlich wie die UML-Objektsichten, eine *Beschreibung von Objekten* darstellen. Bei den hier verwendeten Feature Structures handelt es sich um eine streng typisierte Variante, die auf dem theoretisch fundierten Ansatz von Carpenter ([Carpenter 1992]) basiert. Das klassische Anwendungsgebiet von Feature Structures stellt die KI, speziell die Wissensrepräsentation dar. Da auch Carpenter sein Definitionen auf dieses Gebiet ausrichtet, sind für eine Beschreibung von Objektgraphen in verteilten Systemen durch Feature Structures einige zusätzliche Definitionen zu treffen und Sätze zu beweisen, welche neben Petrinetzen die formale Grundlage für die in Kapitel 4 eingeführten FSNets bilden.

Schließlich wird in Abschnitt 2.4 die Implementierung einer Feature-Structure-Bibliothek dokumentiert, die im Rahmen dieser Arbeit als Voraussetzung für das in Abschnitt 4.5 beschriebene Werkzeug erstellt wurde.

2.1 Klassen und Typen: Grundlage der Informationsmodellierung

Viele Ansätze der Informationsmodellierung legen ein (implizites) Typsystem oder eine Klassenhierarchie zugrunde. Bevor man individuelle Entitäten (Objekte) behandelt, wird damit die Anwendungsdomäne auf abstrakterer Ebene analysiert und modelliert. Beim Übergang von Analyse/Entwurf zur Implementierung können viele der identifizierten Klassen und Typen weiterverwendet werden. Fast alle modernen Programmiersprachen basieren auf einem (statischen oder nur zur Laufzeit verfügbaren) Klassen- oder Typsystem. In Datenbankanwendungen können Spezifikation in

Form von *Entity-Relationship*-Modellen (ER-Modellen) fast direkt in entsprechende Tabellenstrukturen übersetzt werden.

Typen dienen in der Informatik zum einen einer Kategorisierung von Entitäten, zum anderen der Vereinbarung gemeinsamer Eigenschaften wie der zu einem Typ gehörenden Attribute.

Klassen, wie sie als zentrales Konzept der Objektorientierung eingeführt werden (siehe dazu u.a. [Coad und Yourdon 1991, Rumbaugh et al. 1991, Larman 1998]), grenzen sich von Typen nach [Hitz und Kappel 1999], S. 52 ff. durch die folgenden Eigenschaften ab.

- Klassen werden nicht nur als Kategorisierung gleichartiger Objekte verstanden, sondern dienen als Schablonen zur *Erzeugung* von Objekten.
- Ein Objekt stellt während seiner Lebenszeit üblicherweise ein Exemplar *genau einer* Klasse dar, während es zu mehreren Typen gehören kann.
- Klassen spezifizieren neben der auch von Typen angegebenen statischen Sicht zusätzlich das *Verhalten* ihrer Exemplare.
- Klassen können wie Typen in einer Generalisierungsbeziehung stehen. Damit wird bei Klassen aber zusätzlich eine *Codevererbung* impliziert, die Verhalten von der Superklasse auf die Subklasse überträgt.

Diese Eigenschaften sind hier zunächst im Überblick dargestellt, werden im Verlauf dieses Kapitels aber noch näher betrachtet. Aus Sicht der objektorientierten Programmierung hat ein Typ Ähnlichkeit mit dem Konzept der *Schnittstelle* (*interface*, vergleiche [Hitz und Kappel 1999]), das unten ausführlicher beschrieben wird.

Typen und Typrelationen werden in dieser Arbeit aus konzeptueller, aber auch aus formaler Sicht behandelt. UML-Konzepte und -Notationen werden in zwei Schritten eingeführt. Zunächst werden die allgemeinen Techniken zur statischen Informationsmodellierung in UML wie Klassen, Typen, Assoziationen und Generalisierungen in Abschnitt 2.1.1 vorgestellt. Anschließend beginnt der formale Teil dieses Abschnitts mit dem Begriff des Typsystems in Abschnitt 2.1.2, das Typen, Subsumption und Attribut-Zuweisungen behandelt. Abschnitt 2.1.3 zeigt eine besser auf die Modellierung zugeschnittene Formalisierung in Form eines sogenannten Konzeptsystems, das den Möglichkeiten und Notationen von UML näher kommt als das Typsystem, sich aber auf dieses zurückführen läßt. Abschnitt 2.1.4 demonstriert diese Art der Typmodellierung anhand eines Beispiels aus dem Sprachverstehen.

2.1.1 UML-Konzepte und -Notationen für Klassen und Typen

Die Unified Modeling Language (UML) stellt in der Version 1.3 den derzeitigen Standard für objektorientierte Modellierungssprachen dar. Schon deswegen, aber

auch wegen der hohen Akzeptanz in der Praxis, muß sich eine Modellierungstechnik für verteilte Systeme zu UML in Bezug setzen.

Eine der bekanntesten Diagrammarten von UML ist das Klassendiagramm, in der UML-Spezifikation als „statisches Strukturdiagramm“ (*static structure diagram*) bezeichnet, das die statische Sicht auf das Objektmodell beschreibt.

In UML werden Typen als eine spezielle Art von Klassen notiert. Auch zur Darstellung von Typsystemen läßt sich damit das Klassendiagramm nutzen.

Während Klassen in UML als Schablonen für Objekte angesehen werden, beschreiben Typen die Gemeinsamkeiten von Objekten auf einer abstrakteren Ebene. Typisierung von Entitäten besteht normalerweise darin, jeder Entität genau einen Typ zuzuordnen. Es gibt aber auch den allgemeineren Ansatz der Mehrfachtypisierung (siehe z.B. [Fowler und Scott 1997]), bei der jede Entität *mindestens* einen Typ besitzt. Insofern sind Typen ähnlich zu Schnittstellen (*interfaces*), besitzen allerdings im Gegensatz zu diesen durchaus Attribute. Ein Unterschied zwischen Schnittstellen und Typen in UML ist, daß ein Objekt einen Typ dynamisch annehmen und ablegen kann, während seiner gesamten Lebenszeit aber Exemplar ein und derselben Klasse bleibt. Dagegen ist die Zuordnung von Klassen zu den von ihren realisierten Schnittstellen üblicherweise statisch. Der Vergleich von Klassen und Typen wird in [Hitz und Kappel 1999] und [Fowler und Scott 1997] ausführlicher diskutiert. Das dynamischen An- und Ablegen von Typen wird vor allem in bezug auf Rollen genutzt ([Hitz und Kappel 1999], vergleiche als einen aktuellen Beitrag dazu [Papazoglou und Krämer 2000]).

UML sieht für die Darstellung von Typen das Stereotyp *type* vor. *Stereotypen* dienen in UML der Unterscheidung verschiedener Arten eines Modellelements und stellen somit Meta-Typen dar. Sie können textuell in doppelten spitzen Klammern („französische Anführungszeichen“), also beispielsweise als «`type`», oder als für das Stereotyp vereinbartes Piktogramm dargestellt werden. Abbildung 2.1 zeigt ein Beispiel für einen Typ in UML-Darstellung. Eine Klasse wird als Kasten dargestellt, der mehrere *Abschnitte* (*compartments*) enthält.

Im ersten Abschnitt befindet sich der Name der Klasse. Diesem kann ein Stereotyp vorangestellt werden. Es können qualifizierende Attribute als Aufzählung in geschweiften Klammern folgen, beispielsweise {`abstract`} (abstrakte Klasse, siehe z.B. [Larman 1998]) oder {`persistence=persistent`} (Objekte dieser Klasse seien persistent).

Der zweite Abschnitt enthält standardmäßig die Attributdefinitionen. Der dritte Abschnitt mit den *Operationen* der Objekte dieser Klasse wird in dieser Arbeit nicht näher betrachtet, da wir Objekte meist auf Typebene behandeln. In den Abschnitten, welche Implementierungen beschreiben, kommt dieser Abschnitt aber vor. UML unterscheidet zwischen einer Operation, die im Analyse- oder Entwurfs-Klassendiagramm spezifiziert wird, und der *Methode*, die nach UML-Terminologie eine konkrete Implementierung einer Operation darstellt.

Attribute sowie Operationen können eine *Sichtbarkeit* zugeordnet bekommen.

In UML sind hier die aus den Programmiersprachen C++ und Java bekannten Schlüsselworte (in Klammern abkürzende Symbole) `public` (+), `protected` (#) und `private` (-) vorgesehen. Wenn bei Typen keine Sichtbarkeiten angegeben werden, ist wie bei Schnittstellen öffentliche Sichtbarkeit (`public`) gemeint. Es folgt der Name des Attributs, ein Doppelpunkt und der Typ. Der dritte Abschnitt enthält Operationen, die zusätzlich eine Parameterliste spezifizieren. Für Typen werden in dieser Arbeit keine Operationen betrachtet. [Hitz und Kappel 1999] und die UML-Spezifikation ([OMG 2000]) stellen die genaue Syntax und weitere Möglichkeiten ausführlich dar.

Abbildung 2.1 zeigt den Typ Tutor, der ein Attribut bewertung und die Operationen erhöheBewertung() und kannUnterrichten() enthält, die alle öffentlich zugegriffen werden können.

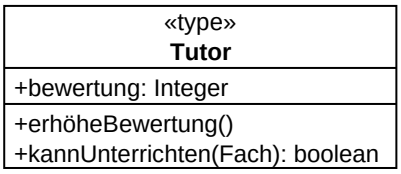

Abbildung 2.1: Darstellung eines Typs in UML.

Da in dieser Arbeit hauptsächlich Typen (nicht Klassen) modelliert werden, soll als Konvention «`type`» die Standardannahme sein. Für den üblichen Standard der Klasse wird hier explizit das Stereotyp «`class`» bzw. für Implementierungsklassen das in [Hitz und Kappel 1999] vorgeschlagene Stereotyp «`implementationClass`», zum Teil abgekürzt als «`impl.Class`», verwendet.

Zwischen Typen existieren im wesentlichen zwei Arten von Relationen: Subtyp-Beziehungen und Assoziationen.

Subtyp-Beziehungen

Eine Subtyp- oder *Ist-Ein*-Relation (is-a) besagt, daß ein Typ einen anderen Typ spezialisiert bzw. verallgemeinert. Dadurch entsteht eine partielle Ordnung (siehe Anhang A, Definition A.3) auf Typen. Als Beispiel sei jede Entität vom Typ Professor auch vom Typ Universitätsangehöriger, so daß Professor als Subtyp von Universitätsangehöriger vereinbart würde („Jeder Professor ist ein Universitätsangehöriger.“).

UML kennt hierfür das allgemeine Konzept der *Generalisierung* zwischen Modellelementen, das auch benutzt wird, um die Subtypbeziehung zwischen Typen auszudrücken. Ein Modellelement wird mit dem Element, von dem es generalisiert wird, durch einen Pfeil mit einer nicht ausgefüllten Pfeilspitze verbunden. Die Pfeilspitze zeigt in UML also zum *allgemeineren* Element. Soll betont werden, daß es

sich bei der Generalisierung um die Subtyp-Beziehung handelt, kann am Generalisierungspfeil das Stereotyp «`isa`» angegeben werden. Da dies sich bei Verwendung von Typen aus dem Kontext ergibt, soll hier darauf verzichtet werden.

Durch Einschränkungen kann eine Generalisierung näher charakterisiert werden. Dafür sieht UML die Schlüsselworte `overlapping` / `disjoint` und `complete` / `incomplete` vor, die nach [Hitz und Kappel 1999] folgende Bedeutung besitzen:

- *overlapping* (überlappend): Die Subtypen sind kompatibel, das heißt ein Objekt kann mehrere dieser Subtypen gleichzeitig annehmen und es lassen sich Subtypen bilden, die von mehreren dieser Subtypen erben.
- *disjoint* (disjunkt): Disjunkte Subtypen sind inkompatibel; weder kann ein Objekt mehrere dieser Subtypen gleichzeitig annehmen, noch ist ein Typ zulässig, der von mehreren dieser Subtypen erbt.
- *complete* (vollständig): Die gegebenen Subtypen stellen alle Möglichkeiten dar, den Typ zu spezialisieren. Dies entspricht der *closed world assumption* (siehe z.B. []) und läßt Schlußfolgerungen vor allem in bezug auf Negation zu.
- *incomplete* (unvollständig): Die gegebenen Subtypen stellen nur die (zur Zeit) bekannten Spezialisierungen des Typs dar. Es kann noch weitere, nicht modellierte Subtypen geben, was dazu führt, daß die *closed world assumption* nicht gilt.

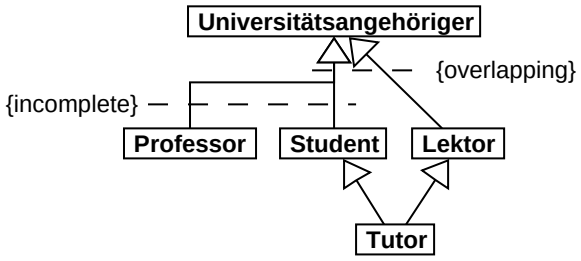

Abbildung 2.2: Verschiedene Arten von Subtypen, dargestellt als Generalisierung in UML-Notation, nach [Hitz und Kappel 1999], S. 43.

Diese Einschränkungen der Subtypisierung werden in Abschnitt 2.1.3 bei der Konstruktion von Typsystemen aus sogenannten Konzeptsystemen wieder aufgegriffen. Abbildung 2.2 zeigt Beispiele für die Darstellung von Subtypen verschiedener Art in UML. Als Standard nimmt UML {`disjunctive,complete`} an. Die gestrichelten Linien zeigen an, auf welche Subtyp-Beziehungen sich die Einschränkungen beziehen. So wird ausgedrückt, daß Professor und Student nicht alle möglichen Subtypen von Universitätsangehöriger sind (`incomplete`), aber disjunkt („Niemand ist

gleichzeitig Professor und Student“). Durch die Modellierung des Subtyps Lektor als `overlapping` zu den anderen Subtyp-Beziehungen wird ausgedrückt, daß sowohl Professoren als auch Studenten Lektoren sein können. Dadurch ist der von Student und Lektor erbende Subtyp Tutor möglich, der für studentische Lektoren einen expliziten Typ zur Verfügung stellt.

Als weitere Notationskonvention soll die Unterscheidung zwischen `overlapping` und `disjunctive` im folgenden graphisch dargestellt werden. Während in UML das Zusammenführen mehrerer Subtyp-Beziehungen zu einer Pfeilspitze keine semantische Bedeutung besitzt, soll dies hier, wie in Abbildung 2.2 bereits angedeutet, die explizite Beschriftung mit den Einschränkungen ersetzen. In einer Pfeilspitze zusammenlaufende Subtyp-Beziehungen seien also `disjunctive`, alle Subtyp-Beziehungen mit eigener Pfeilspitze dagegen `overlapping`.

Da in dieser Arbeit keine Negation von Typen betrachtet wird, ist der Unterschied zwischen `complete` und `incomplete` unerheblich und wurde hier nur der Vollständig halber erwähnt.

Assoziationen

Assoziationen zwischen Typen oder Klassen werden in verschiedene Arten unterteilt und verfügen über weitere Eigenschaften. Es gibt Assoziationen verschiedener Stelligkeit, wobei die zweistellige Assoziation den Normalfall darstellt. Eine Assoziation bekommt genau wie ein Typ einen Namen zugeordnet. UML sieht zusätzlich vor, daß jedes Assoziations*ende* einen Namen bekommt, der die Rolle der Klasse in der Assoziation beschreibt. Wird hier kein eigener Name angegeben, wird als Standard für den Namen des Assoziationsendes der Name der mit diesem verbundenen Klasse oder bei gerichteten Assoziationen der Name der Assoziation selbst angenommen. Bei Assoziationen spricht man in UML weiterhin von Multiplizitäten, Navigierbarkeit, abgeleiteten Assoziationen und speziellen Assoziationsarten wie Aggregation und Komposition.

Multiplizitäten (*multiplicities*) werden für beide Assoziationsenden angegeben und unterscheiden zum einen zwischen *kann*- und *muß*-Beziehung, zum anderen zwischen einfacher und mehrfacher Wertigkeit der Assoziation. Eine *kann*-Beziehung wird durch die Multiplizität `0..`n ausgedrückt, während eine *muß*-Beziehung entsprechend eine Mindestmultiplizität von Eins vorsieht (`1..`n). Der Wert von n hängt wiederum von der Wertigkeit des Assoziationsendes ab: Bei $n =$`1` handelt es sich um eine einfache, sonst um eine mehrfache Wertigkeit. Einen häufig anzutreffenden Sonderfall für n stellt die Notation `*` dar, die eine unbeschränkte mehrfache Wertigkeit spezifiziert.

Die *Navigierbarkeit* (*navigability*) von Assoziationen betrifft die Frage, ob die assoziierten Objekte sich gegenseitig kennen (beidseitige oder symmetrische Assoziation) oder nur eins das andere, aber nicht umgekehrt (gerichtete Assoziation).

Abgeleitete Assoziationen und Attribute (*derived associations / attributes*) erge-

ben sich durch Berechnungen aus anderen Assoziationen und Attributen. Sie werden in der Realisierung meist nicht durch entsprechende Exemplarvariablen dargestellt, sondern aus Methodenaufrufen oder anderen Exemplarvariablen hergeleitet bzw. berechnet. Wenn für abgeleitete Attribute doch eine Exemplarvariable genutzt wird, so zur effizienten Zwischenspeicherung des Berechnungsergebnisses (*caching*). Notiert werden abgeleitete Assoziationen oder Attribute, indem dem Namen ein Schrägstrich vorangestellt wird.

Attribute und Assoziationen können wie alle UML-Modellelemente *Eigenschaften* (*properties*) zugeordnet bekommen. Da diese Eigenschaften Attribute der Modellelemente darstellen, werden sie auch als *Meta-Attribute* bezeichnet. Beispiele für solche Meta-Attribute von Attributen sind {`readOnly`}, das nur lesenden Zugriff auf das Attribut erlaubt und {`frozen`}, das nur eine einmalige Zuweisung an ein Attribut erlaubt und damit dem Modifikator `final` in Java entspricht.

Assoziationsarten wie *Aggregation* und *Komposition* sollen Spezialfälle der Beziehungen zwischen Objekten konzeptualisieren, die häufig unter dem Begriff *Teil-Ganzes*-Beziehung zusammengefaßt werden. Die Details sind subtil und werden z.B. in [Hitz und Kappel 1999] ausführlich diskutiert. Beide werden durch eine Raute am Assoziationsende des aggregierenden / komponierenden Objekts dargestellt, wobei diese für Kompositionen ausgefüllt wird.

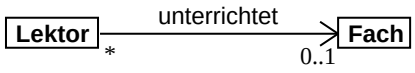

Abbildung 2.3: Darstellung einer Assoziation in UML-Notation.

Attribute können als eine gerichtete, einwertige und benannte *kann*-Assoziationen zwischen Typen aufgefaßt werden, die oft auch als Aggregation oder Komposition interpretiert wird. Der Name des Attributs entspricht dabei dem Rollennamen des Zieltyps oder (wenn sich hier kein sinnvoller Rollenname finden läßt) dem Namen der gesamten Assoziation. Der Unterschied zwischen einem Attribut und einer solchen Assoziation ist gering und eher Geschmackssache, als formal festzuhalten. Oft werden Attribute in der Objektorientierung nur für Assoziationen zu Basiswerten wie Zahlen oder Zeichenketten eingesetzt, wenn diese nicht als echte Objekte (*first class objects*) aufgefaßt werden. Das Beispiel in Abbildung 2.3 zeigt eine Attribut-Assoziation, die den Typ Lektor mit dem Typ Fach über das Attribut unterrichtet verbindet („Ein Lektor kann ein Fach unterrichten.“). Der Stern über dem linken Assoziationsende scheint bei gerichteten Assoziationen weniger wichtig, da in diese Richtung nicht navigiert werden kann. Er beschreibt allerdings dennoch eine wichtige semantische Unterscheidung, nämlich im Beispiel, daß ein Fach in mehreren Assoziationen auftreten kann, also von verschiedenen Lehrer-Objekten referenziert (unterrichtet) werden darf. Diese Situation ist beispielsweise bei Kompo-

sitionen nicht erlaubt, so daß in einem solchen Fall das linke Assoziationsende mit `1` beschriftet würde. Da im folgenden hauptsächlich einwertige und als Werte beliebig oft wiederverwendbare *kann*-Attribute betrachtet werden, sei die Standardannahme, daß das Quell-Assoziationsende die Multiplizität `*` und das Ziel-Assoziationsende `0..1` besitzen. Bei der Assoziation in Abbildung 2.3 wären also unter diesen Annahmen keine Multiplizitätsangaben erforderlich.

Klassen und Typen werden in UML wie alle Modellelemente in *Namensräumen* (*name spaces*) organisiert, wobei speziell für Klassen *Pakete* vorgesehen sind, welche einen Namensraum für eine Menge von Klassen und weiteren (Sub-)Paketen definieren. Der Namensraumbezeichner kann Klassen- oder Typnamen zur eindeutigen Referenz mit dem Trennsymbol `::` vorangestellt werden.

Ein Beispiel für ein UML-Klassendiagramm

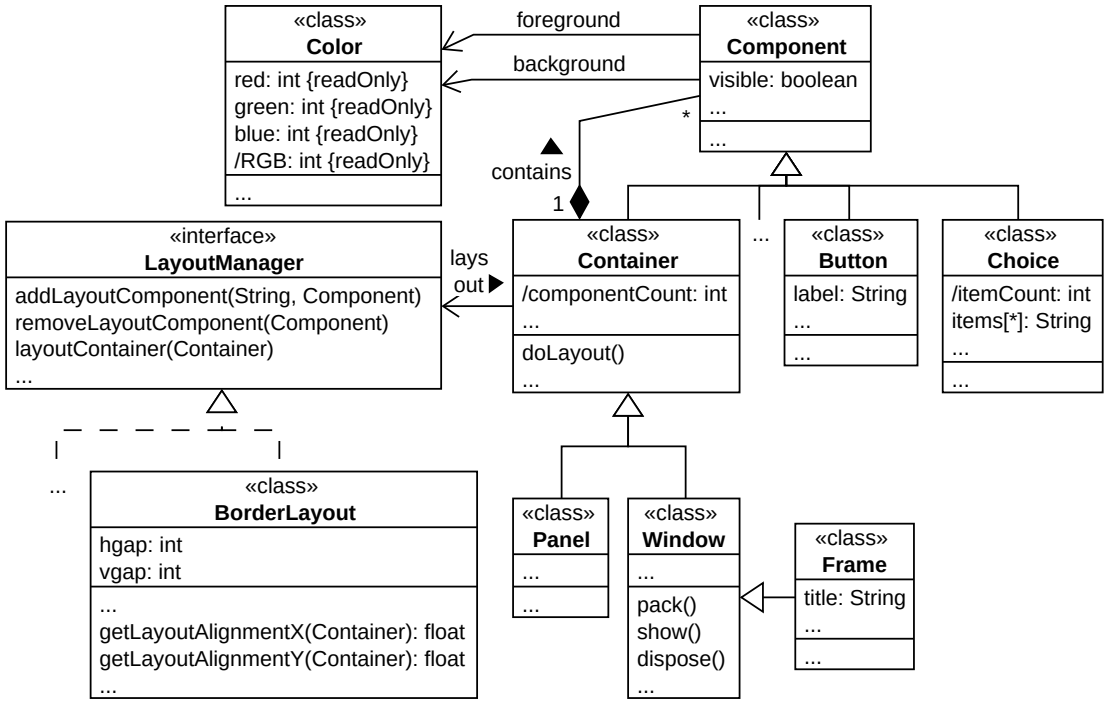

Abbildung 2.4: Ein Ausschnitt des statischen Modells des *Abstract Windowing Toolkit* von Java als UML-Klassendiagramm.

Es wird nun ein etwas komplexeres Klassendiagramms als Beispiel erläutert. Abbildung 2.4 zeigt einen Ausschnitt des statischen Modells des *Abstract Windowing Toolkit* (AWT) von Java (siehe [Sun 2000b], [Gosling et al. 1996]) als UML-Klassendiagramm.

Auch in Java gibt es das Konzept der Pakete ([Arnold und Gosling 1996]); hier ist als Trennsymbol statt `::` ein Punkt üblich. Sämtliche in der Abbildung verwendeten Klassen stammen aus dem Paket `java.awt`, weshalb der Paketname nicht angegeben wird. Fehlende Attribute oder Methoden und weitere Subklassen sind jeweils durch drei Punkte angedeutet.

Für die Klassen wurde das Stereotyp «`class`» verwendet, da die Klassen gegenüber ihrer Implementierung («`impl.Class`») abstrahiert dargestellt sind. Dies bezieht sich vor allem auf die Darstellung von Attributen und Assoziationen, die in der Implementierung nicht als öffentliche Attribute, sondern nach einem Entwurfsmuster für Attribute durch `get-` und `set`-Methoden realisiert werden. Als {`readOnly`} werden Attribute dargestellt, die eine öffentliche `get-`, aber keine oder eine nicht-öffentliche `set`-Methode anbieten. Für mehrwertige Attribute oder Assoziationen gibt es in Java ein ähnliches Entwurfsmuster, in dem die Methoden `add...(...)`, `remove...(...)` und `...Count():int` verwendet werden, um Objekte hinzuzufügen, zu entfernen und die aktuelle Kardinalität der Assoziation abzufragen. Die Kardinalität ist im Beispiel durch abgeleitete Attribute dargestellt, da sich deren Wert aus der Assoziation oder dem mehrwertigen Attribut ableiten läßt. Vererbte Attribute und Assoziationen sind in der Abbildung nicht nochmals angegeben. Dies kann aber in Entwurfs-Klassendiagrammen geschehen, um ein *Überschreiben* hervorzuheben.

Das Java-AWT erlaubt die plattformunabhängige Definition von graphischen Benutzungsschnittstellen (*graphical user interfaces*, GUIs). Es stellt ein Objektmodell der GUI zur Verfügung, das auf den zentralen Konzepten Komponente (Klasse Component) und Behälter (Container) aufbaut. Komponenten sind für den Benutzer sichtbare Teile der GUI, die durch Behälter zu Gruppen zusammengefaßt werden. Neben grundlegenden Datenobjekten wie Farben (Color), Punkten, Dimensionen und Rechtecken (im Beispiel nicht dargestellt) gibt es diverse solche Komponenten wie Schaltflächen (Button) und Auswahlfelder (Choice) und verschiedene Behälter, von denen Fenster (Window) und Fläche (Panel) die wichtigsten darstellen.

Die in Abbildung 2.4 dargestellte `contains`-Komposition zwischen Behälter und Komponenten spezifiziert das Enthalten-Sein von vielen Komponenten (mehrwertige Assoziation) in einem Behälter. Sie vererbt sich auf alle Subklassen von `Component`, so daß ein Behälter Objekte beliebiger Komponenten-Klassen enthalten kann. Durch die Vererbungsbeziehung zwischen Behälter und Komponente ist jeder Behälter auch eine Komponente. Damit ist eine hierarchische Schachtelung von Behältern möglich.

Ein Behälter besitzt weiterhin einen `LayoutManager`, der für die Anordnung der Komponenten des Behälters zuständig ist. `LayoutManager` selbst ist im AWT als Schnittstelle realisiert, die von verschiedenen Klassen implementiert wird. `BorderLayout` ist eine einfache Realisierung der Schnittstelle, die eine Anordnung von maximal fünf Komponenten an den vier Rändern und in der Mitte des Behälter-Rechtecks erlaubt.

UML-Klassendiagramme stellen einen Standard zur Darstellung von Typen,

Klassen, Schnittstellen und den Beziehungen zwischen ihnen dar. Von den vielen Diagrammtypen in UML zählt das Klassendiagramm zu den ausgereiftesten, was durch die lange Tradition in der Datentyp- und Entity-Relationship-Modellierung zu erklären ist ([Papazoglou et al. 2000]). Mit etwas Übung sind Klassendiagramme deshalb vor allem für Programmierer leicht zu lesen und zu verstehen. Kritisiert wird an Klassendiagrammen, daß sie gerade in einer frühen Phase der Modellierung durch ihren Detailreichtum und ihre an Programmiersprachen orientierten Konzepte zu einem implementierungsorientierten Herangehen verleiten ([Griffel 1998]). Als Alternativen zur Modellierung statischer Strukturen werden semantische Netze (siehe bei [Merz et al. 1999a]) und erweiterte ER-Diagramme ([Papazoglou 1995]) vorgeschlagen, bei denen der Modellierer zu einer abstrakteren Sichtweise gezwungen wird.

Die Konzepte und Notationen von UML werden im folgenden Abschnitt zu einer formalen Definition von Typen und Typrelationen in bezug gesetzt. Die resultierende Typmodellierung stellt sich als Teilmenge von UML dar, die sich auf die reinen Informationsaspekte konzentriert und damit eine abstraktere Modellierung erlaubt. Sämtliche Aspekte des Systemverhaltens, und damit auch die statischen Aspekte, die in UML in Form von Operationen im Klassendiagramm auftreten, werden dagegen im hier vorgestellten Ansatz in den Bereich der Prozeßmodellierung verlagert.

2.1.2 Formalisierung von Typen und Subsumption

Für die im vorherigen Abschnitt informal eingeführten Typen und Typbeziehungen sollen in diesem Abschnitt formale Definitionen gegeben werden, die sich nach [Carpenter 1992] richten.

Eine Subtyp-Beziehung über einer Menge Type von Typen wird dort als eine partielle Ordnung $(\sqsubseteq) \subseteq \text{Type} \times \text{Type}$ dargestellt, welche die Typen nach Spezialisierung sortiert. Es wird deshalb eine *partielle* Ordnung gewählt, um die Spezifikation unvergleichbarer Typen zu ermöglichen (weder subsumiert ein Typ den anderen noch umgekehrt) sowie zyklische Abhängigkeiten, also Vererbungsschleifen, auszuschließen.

Definition 2.1 [Typ-Subsumptionsrelation]
Eine *Typ-Subsumptionsrelation* $\sqsubseteq$ über einer Menge Type von Typen ist eine abgeschlossene partielle Ordnung $(\sqsubseteq) \subseteq \text{Type} \times \text{Type}$ nach Def. A.10. ◊

Für $\sigma \sqsubseteq \tau, \sigma, \tau \in \text{Type}$ sagt man auch, σ subsumiert τ, ist allgemeiner als oder Supertyp von τ. Von rechts nach links gelesen bedeutet die Aussage τ wird subsumiert von, ist spezieller als oder ist Subtyp von σ.

Eine wichtige Frage ist die nach der Konsistenz von Typen. Diese läßt sich zurückführen auf den im Anhang unter Definition A.6 eingeführten Begriff der konsistenten Menge bezüglich einer partiellen Ordnung. Eine Menge von Typen $\mathrm{T} \subseteq \text{Type}$ heißt konsistent, wenn alle Elemente $\tau \in \mathrm{T}$ einen gemeinsamen Subtyp $\sigma \in \text{Type}$

besitzen, also wenn für alle $\tau \in \mathrm{T}$ gilt $\tau \sqsubseteq \sigma$ (siehe Definition A.6). Um bei der Unifikation von Typen eindeutige Ergebnisse zu bekommen, wird gefordert, daß für jede konsistente Teilmenge von Type eine solche kleinste obere Schranke eindeutig definiert ist (siehe Definition A.9), was zum Begriff der abgeschlossenen partiellen Ordnung führt. Die Typ-Unifikation oder der *Join* liefert für jede konsistente Teilmenge T von Type den allgemeinsten Typ, der spezieller als alle in T enthaltenen Typen ist.

Definition 2.2 [Join]
Sei Type eine Menge von Typen und $\sqsubseteq$ eine Typ-Subsumptionsrelation und damit $\langle Type, \sqsubseteq\rangle$ eine partiell geordnete Menge nach Definition A.4. Der Join $\bigsqcup$ einer Menge $\mathrm{T} \subseteq$ Type wird definiert als das Supremum von T bezüglich $\langle Type, \sqsubseteq\rangle$, also $(\bigsqcup) := \sup_{\langle \mathrm{Type}, \sqsubseteq\rangle}$ (siehe Definition A.9).
Für zweielementige Mengen wird auch die Infixnotation $\sigma \sqcup \tau := \bigsqcup\{\sigma, \tau\},\ \sigma, \tau \in$ Type verwendet. ◊

Da eine Typ-Subsumptionsrelation als *abgeschlossene* partielle Ordnung definiert wurde, ist der Join einer Typmenge immer *eindeutig* definiert oder undefiniert. Daraus folgt, daß es immer einen allgemeinsten Typ, genannt *top* (symbolisch $\top$), geben muß, den man erhält, wenn man den Join über die leere Menge ($\top = \bigsqcup\emptyset$) bildet. Dieser Typ subsumiert alle anderen Typen aus Type und übernimmt damit die Rolle des aus objektorientierten Programmiersprachen bekannten Typs Object.

Typen werden weiterhin genutzt, um einheitlich zu definieren, welche Attribute in Entitäten erlaubt sind und wiederum für jedes Attribut, welchen Typ dieses für seinen Wert fordert. Carpenter führt hierfür eine Funktion *Approp* ein, die als Argumente einen Typ und ein Attribut erhält und den *angemessenen* (*appropriate*) Typ liefert, den eine Entität mindestens haben muß, damit sie dem Attribut als Wert zugewiesen werden darf.

Statt von Attributen wird hier im folgenden der englische Begriff *Feature* benutzt, wenn der Unterschied zwischen Attributen, wie sie in UML vorkommen, und durch Typen definierte Features hervorgehoben werden soll. Der Begriff soll weiterhin den Bezug zu Feature Structures verdeutlichen, zu deren Typisierung in Abschnitt 2.3.4 die hier definierten Typen verwendet werden.

Feature-Typisierungen vererben sich durch die Typhierarchie. Dies bedeutet, daß ein Feature, das in einem Typ erlaubt ist, auch in allen Subtypen erlaubt ist, und diese den Wertebereich für das Feature weiter einschränken (aber nicht lockern) dürfen. Dies entspricht der Kovarianz und damit dem Substitutionalitätsprinzip der Objektorientierung, das besagt, daß ein Objekt eines spezielleren Typs überall dort einsetzbar sein muß, wo ein Objekt eines allgemeineren Typs gefordert ist. Mit den hier verwendeten Begriffen und Notationen sei eine Feature-Typisierung folgendermaßen definiert.

Definition 2.3 [Feature-Typisierung]
Eine *Feature-Typisierung* approp über einer Typmenge Type mit Subsumptionsre-

lation $\sqsubseteq$ und einer Featuremenge Feat ist eine partielle Funktion approp : Type $\times$ Feat $\hookrightarrow$ Type, für die für alle $\sigma, \tau \in$ Type, $f \in$ Feat gilt: Wenn approp(σ, f) definiert ist und $\sigma \sqsubseteq \tau$, dann ist approp(τ, f) auch definiert und approp$(\sigma, f) \sqsubseteq$ approp(τ, f). $\Diamond$

Um die in der UML-Notation in Abschnitt 2.1.1 eingeführten mehrwertigen Attribute und Assoziationen darstellen zu können, wird in dieser Arbeit auf Listen zurückgegriffen, die zusammen mit einer speziellen Notation in Abschnitt 2.3.5 ausführlich vorgestellt werden.

Der Begriff des Typsystems faßt eine Typmenge, eine zugehörige Subsumptionsrelation, eine Featuremenge und eine Feature-Typisierung zu einem (mathematischen) Objekt zusammen. Hier fordern wir, daß Typ- und Featuremenge endlich sind.

Definition 2.4 [Typsystem]
Ein Typsystem ist ein Tupel
TS $= \langle$Type, $\sqsubseteq$, Feat, approp$\rangle$, wobei

(a) Type eine endliche Menge von Typen,

(b) $\sqsubseteq$ eine abgeschlossene Typ-Subsumptionsrelation über Type nach Def. A.10,

(c) Feat eine endliche Menge von Features und

(d) approp eine Feature-Typisierung über Type, $\sqsubseteq$ und Feat nach Def. 2.3 ist. $\Diamond$

Damit ist ein einfaches, aber wie sich in Kapitel 5 zeigen wird, in der praktischen Modellierung hinreichend ausdrucksstarkes Typsystem mathematisch definiert. Im folgenden Abschnitt leiten wir eine Darstellung von Typsystemen her, die sich besser zu deren Spezifikation eignet.

2.1.3 Typsysteme aus Konzeptsystemen

Im vorherigen Abschnitt wurde der Begriff des Typsystems als ein Tupel aus Typen, Subsumptionsrelation, Features und Feature-Typisierung eingeführt. Sowohl für die Subsumptionsrelation als auch die Feature-Typisierung wurden bestimmte Einschränkungen gefordert, um zu einem wohldefiniertem Typsystem zu gelangen.

Betrachten wir nun, wie ein Typsystem von einem Modellierer spezifiziert werden kann. Ein Software-Werkzeug könnte zu einem gegebenen Tupel, das syntaktisch einem Typsystem nach Definition 2.4 entspricht, entscheiden, ob dieses die dort angegebenen semantischen Einschränkungen einhält und somit ein wohldefiniertes Typsystem darstellt. Das Typsystem stellt aber eine nur aus mathematischer Sicht gut handhabbare Repräsentation dar, nicht eine aus Modellierersicht gut verwendbare. Selbst wenn das Software-Werkzeug gut verständliche Fehlermeldungen generiert, müßte der Modellierer selbst für korrekte Vererbungshierarchien (wie Transitivität,

Fehlen von Vererbungsschleifen), richtige Feature-Vererbung (Auftreten von Features in Subtypen, zugewiesener Typ von Subtyp-Features) und anderes sorgen. Es wird also eine Repräsentation gesucht, die näher an den Vorstellungen des Modellierers ist und viele der genannten Fehler gar nicht erst erlaubt, sondern die meisten Eigenschaften eines Typsystems automatisch sicherstellt.

Der hier vorgeschlagene Ansatz basiert auf zwei Grundlagen: Auf den Vorschlägen aus [Carpenter 1992] und auf der heutigen Standard-Modellierungssprache der Objektorientierung, der bereits häufiger erwähnten Unified Modeling Language (UML).

Aus [Carpenter 1992] stammt der Vorschlag, Typhierarchien aus sogenannten ISA-*Netzwerken* (ISA *networks*) zu konstruieren. Da Carpenter bei dem Basiselement, aus dem Typen konstruiert werden, von einem *Konzept* (*concept*) spricht, werden ISA *networks* in dieser Arbeit zum Begriff des *Konzeptsystems* erweitert.

Carpenter betrachtet die disjunktive und die konjunktive Konstruktion von Typen aus ISA-Netzwerken, wobei hier nur die konjunktive Variante herangezogen wird, da diese besser den Darstellungen aus UML und den Vorschlägen zur Mehrfachtypisierung (siehe folgender Abschnitt) entspricht. Weiterhin eignet sich die konjunktive Darstellung besonders für einen einfacheren Typ-Unifikationsalgorithmus.

Ein ISA-Netzwerk ist nach Carpenter nichts anderes als eine endliche partiell geordnete Menge (siehe Anhang A.4), wobei diese im Gegensatz zu einer Subsumptionsrelation nicht abgeschlossen sein muß (vergleiche Definition 2.1). Es besteht aus einer Menge von Konzepten und einer partiellen Ordnung über dieser Menge, die ISA-Relation genannt wird. Genau wie in einer Subsumptionsbeziehung sind die Eigenschaften Reflexivität, Transitivität und Anti-Symmetrie, die für eine partielle Ordnung gefordert werden, für eine ISA-Relation sinnvoll: Sie sorgen für eine angemessene Vererbungssemantik und verhindern insbesondere Vererbungsschleifen. Vererbungsschleifen führen dazu, daß für zwei unterschiedliche Konzepte k und k' gilt, daß k ISA k' und k' ISA k. Dadurch können k und k' in bezug auf ihren Informationsgehalt nicht unterschieden werden und sollten deshalb als ein gemeinsames Konzept modelliert werden.

Ein ISA-Netzwerk kann in dieser Form ausdrücken, welche Konzepte spezieller als andere sind, nicht aber, welche Konzepte inkompatibel sind. Bei Typen wird die Inkompatibilität dadurch ausgedrückt, daß kein gemeinsamer Subtyp existiert. Um anzugeben, welche Typen konstruiert werden sollen, also welche konjunktiven Verbindungen von Konzepten möglich sein sollen, ist es nötig, auf Konzeptebene Inkompatibilität explizit zu spezifizieren. Dafür dient im Vorschlag von Carpenter ([Carpenter und Thomason 1990]) die ISNOTA-Relation über die Menge der Konzepte Conc, die sinnvollerweise für alle $p, q, r \in \text{Conc}$ folgende Eigenschaften besitzt:

- p ISNOTA $q \iff q$ ISNOTA p (Symmetrie)
- $\neg\ p$ ISNOTA p (Anti-Reflexivität)
- p ISA $q \wedge q$ ISNOTA $r \implies p$ ISNOTA r (Verkettung)

Daraus ergibt sich die Definition eines IS(NOT)A-Netzwerks wie folgt.

Definition 2.5 [IS(NOT)A**-Netzwerk**]
Ein IS(NOT)A-Netzwerk ist ein Tupel $\langle \text{Conc}, \text{ISA}, \text{ISNOTA} \rangle$, wobei

(a) Conc eine endliche Menge von Konzepten ist,

(b) ISA eine partielle Ordnung über Conc darstellt und

(c) ISNOTA eine symmetrische und anti-reflexive Relation über Conc ist, für die gilt:
$\forall p, q \in \text{Conc} : p \text{ ISA } q \wedge q \text{ ISNOTA } r \Rightarrow p \text{ ISNOTA } r.$ ♢

Typen werden nun aus Mengen von Konzepten konstruiert, wobei zwei unterschiedliche Konzepte eines Typs weder in der ISA-, noch in der ISNOTA-Beziehung stehen dürfen, da im ersten Fall das allgemeinere nicht zur Aussagekraft des Typs beitrüge (Redundanz) und im zweiten Fall inkompatible Konzepte gemeinsam auftreten würden (Inkonsistenz). Mit einer konjunktiven Typkonstruktion wird die Menge der Typen folgendermaßen aus einem IS(NOT)A-Netzwerk erstellt ([Carpenter 1992]):

Definition 2.6 [**Konstruktion konjunktiver Typen**]
Sei $\text{N} = \langle \text{Conc}, \text{ISA}, \text{ISNOTA} \rangle$ ein IS(NOT)A-Netzwerk. Die aus N hergeleitete Menge Type der *konjunktiven Typen* sei so definiert, daß $\sigma \in \text{Type}$ genau dann wenn:

(a) $\sigma \subseteq \text{Conc}$,

(b) $\neg\exists p, q \in \sigma, p \neq q : p \text{ ISA } q$ und

(c) $\neg\exists p, q \in \sigma : p \text{ ISNOTA } q$ ♢

Die Subsumptionsrelation über den konjunktiv konstruierten Typen ergibt sich nach [Carpenter 1992] folgendermaßen.

Definition 2.7 [**Subsumption konjunktiver Typen**]
Sei $N = \langle \text{Conc}, \text{ISA}, \text{ISNOTA} \rangle$ ein IS(NOT)A-Netzwerk und Type die nach Definition 2.6 daraus konstruierte Typmenge. Die aus N hergeleitete Subsumptionsrelation $\sqsubseteq$ sei definiert als:

$$forall\sigma, \tau \in \text{Type} : \sigma \sqsubseteq \tau \Longleftrightarrow \forall s \in \sigma \; \exists t \in \tau : t \text{ ISA } s$$ ♢

Die Subsumption zwischen Typen läßt sich also auf die ISA-Relation zwischen Konzepten zurückführen, indem jedes Konzept des allgemeineren Typs auf ein Konzept des spezielleren Typs abgebildet wird, das selbst wiederum spezieller ist. Ein aus Konzepten konjunktiv konstruierter Typ kann damit nach Definition 2.7 auf verschiedene Weisen spezialisiert werden:

- indem Konzepte hinzugefügt werden. Die Abbildung bleibt dieselbe, die hinzugefügten Konzepte sind nur nicht als Bilder enthalten.

- indem einzelne Konzepte spezialisiert werden. Wiederum bleibt die Abbildung dieselbe, t ISA s gilt weiterhin.
- indem mehrere Konzepte zu einem spezielleren zusammengefaßt werden. In diesem Fall werden mehrere Konzepte des allgemeineren Typs auf dasselbe Konzept des spezielleren Typs abgebildet. In der Konzepthierarchie tritt eine Mehrfachvererbung auf.

Da die Spezialisierung transitiv ist, können diese Regeln natürlich beliebig häufig angewandt und kombiniert werden.

Carpenter zeigt, daß durch die gegebene Konstruktion tatsächlich eine wohldefinierte Subsumptionsrelation über die konstruierte Typmenge erstellt wird. Dafür zeigt er, daß die Relation $\sqsubseteq$ eine *abgeschlossene* partielle Ordnung darstellt. Weiterhin beweist er, daß die folgende spezielle Definition des Join bei konjunktiven Typen der zuvor gegebenen entspricht.

Definition 2.8 [Konstruktion des Join konjunktiver Typen]
Sei $N = \langle \text{Conc}, \text{ISA}, \text{ISNOTA} \rangle$ ein IS(NOT)A-Netzwerk, Type die nach Definition 2.6 konstruierte Typmenge und $\sqsubseteq$ die in Definition 2.7 gegebene Subsumptionsrelation. Der Join zweier Typen $\sigma, \tau \in \text{Type}$ ergibt sich nach [Carpenter 1992], S. 27, als

$$\sigma \sqcup \tau := \begin{cases} \max_{\langle \text{Conc,ISA} \rangle}(\sigma \cup \tau) & \text{falls } \neg\exists p \in \sigma, q \in \tau : p \text{ ISNOTA } q \\ \text{undefiniert} & \text{sonst} \end{cases} \qquad \Diamond$$

Der Join von konjunktiven Typen kann also einfach durch Vereinigung der als Konzeptmengen dargestellten Typen berechnet werden, falls diese Vereinigung bezüglich ISA konsistent ist. Dabei werden durch die im Anhang A, Definition A.7 definierte Funktion max Konzepte, die durch andere Konzepte der Vereinigung spezialisiert werden, herausgefiltert. Wichtig ist, daß im *sonst*-Fall der Join undefiniert ist, nicht etwa die leere Menge, da diese dem Typ *top* ($\top$) entspricht.

Auch eine Feature-Typisierung soll auf Konzeptebene spezifizierbar sein, um diese auf die Typebene übertragen zu können. Für diese Art der Feature-Typisierung wird analog zur Typebene gefordert, daß ein Subkonzept den Typ eines Features nur so ändern darf, daß dieser spezieller wird.

Definition 2.9 [Konzept-Feature-Typisierung]
Sei $N = \langle \text{Conc}, \text{ISA}, \text{ISNOTA} \rangle$ ein IS(NOT)A-Netzwerk, Type die nach Definition 2.6 konstruierte Typmenge und $\sqsubseteq$ die in Definition 2.7 gegebene Subsumptionsrelation. Eine *Konzept-Feature-Typisierung* approp_c über N und einer Featuremenge Feat ist eine partielle Funktion $\text{approp}_\text{c} : \text{Conc} \times \text{Feat} \hookrightarrow \text{Type}$, für die für alle $p, q \in \text{Conc}$, $f \in \text{Feat}$ gilt:

(a) Wenn $\text{approp}_\text{c}(p, f)$ definiert ist und p ISA q, dann ist $\text{approp}_\text{c}(q, f)$ auch definiert und $\text{approp}_\text{c}(p, f) \sqsubseteq \text{approp}_\text{c}(q, f)$.

(b) Wenn $\text{approp}_c(p, f)$ und $\text{approp}_c(q, f)$ definiert sind, aber $\text{approp}_c(p, f) \sqcup \text{approp}_c(q, f)$ undefiniert ist, so muß gelten p ISNOTA q. ◊

Während die erste Bedingung für die auch auf der Typebene definierte Feature-Vererbung sorgt, verhindert die zweite einen subtilen Grund für Inkonsistenzen zwischen Typen, die sonst durch die Feature-Typisierung entstehen könnten. Wenn zwei Konzepte eines Typs für dasselbe Feature inkompatible Typen verlangen, kann für den Typ keine konsistente Feature-Typisierung für dieses Feature gefunden werden. Deswegen sorgt die zweite Bedingung dafür, daß solche Konzepte von vornherein durch die ISNOTA-Relation als inkompatibel gekennzeichnet sein müssen.

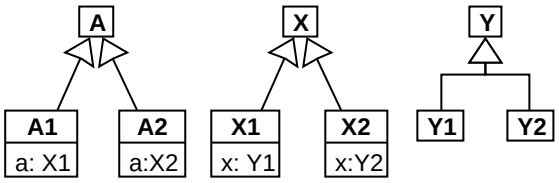

Abbildung 2.5: Mehrfach-indirekte Inkonsistenz durch Feature-Typisierung.

Ein Beispiel für eine solche indirekte Inkonsistenz zeigt Abbildung 2.5, wobei die ISNOTA-Relation wie in der Notationskonvention in Abschnitt 2.1.1 vereinbart durch disjunkte Subtypen gegeben ist. Im Beispiel besitzt der Typ A die Subtypen A1 und A2. Es scheint zunächst so, als seien diese Subtypen kompatibel, demnach wäre A1, A2 ein zulässiger Typ. Dasselbe gilt für X mit den Subtypen X1 und X2, wogegen die Subtypen Y1 und Y2 explizit als disjunkte und damit inkompatible Subtypen von Y gekennzeichnet sind. Durch die Feature-Typisierung der X-Subtypen ergibt sich implizit, daß auch diese inkompatibel sein müssen, da es wegen der Inkompatibilität von Y1 und Y2 keinen Wert für das Feature x geben kann. Das gleiche gilt für die A-Subtypen und das Feature a, so daß eine über zwei Ebenen indirekte Inkonsistenz entsteht.

Aus der oben definierten Konzept-Feature-Typisierung soll nun eine Feature-Typisierung für die konjunktiv konstruierten Typen abgeleitet werden.

Definition 2.10 [Konstruktion der Feature-Typisierung konjunktiver Typen]
Sei $N = \langle \text{Conc}, \text{ISA}, \text{ISNOTA} \rangle$ ein IS(NOT)A-Netzwerk, Type die nach Definition 2.6 konstruierte Typmenge, $\sqsubseteq$ die in Definition 2.7 gegebene Subsumptionsrelation und approp_c eine Konzept-Feature-Typisierung über einer Featuremenge Feat. Sei für alle Typen $\sigma \in$ Type und Features $f \in$ Feat die Typmenge $C_{\sigma,f}$ die Menge aller dem Feature f durch alle Konzepte von σ zugeordneten Typen, definiert als $C_{\sigma,f} := \{\text{approp}_c(s, f) \mid s \in \sigma\}$. Die approp_c zugeordnete Feature-Typisierung approp zu

Type sei dann definiert als

$$\text{approp}(\sigma, f) := \begin{cases} \bigsqcup C_{\sigma,f} & \text{falls } C_{\sigma,f} \neq \emptyset \\ \text{undefiniert} & \text{sonst} \end{cases} \quad \Diamond$$

Um analog zum Typsystem die Konzepte, Features und die dazugehörigen Typisierungen als ein Objekt zu erhalten, sei der Begriff des Konzeptsystems eingeführt. Ein Konzeptsystem enthält neben dem IS(NOT)A-Netzwerk eine Featuremenge und eine Feature-Typisierung auf Konzeptebene.

Definition 2.11 [Konzeptsystem]
Ein *Konzeptsystem* ist ein Tupel $\text{CS} = \langle \text{Conc}, \text{ISA}, \text{ISNOTA}, \text{Feat}, \text{approp}_\text{c} \rangle$, wobei

(a) $N = \langle \text{Conc}, \text{ISA}, \text{ISNOTA} \rangle$ ein IS(NOT)A-Netzwerk,

(b) Feat eine Menge von Features und

(c) approp_c eine Konzept-Feature-Typisierung über N ist. $\Diamond$

Aus einem solchen Konzeptsystem kann nun ein Typsystem abgeleitet werden.

Definition 2.12 [Konstruktion eines Typsystems aus eines Konzeptsystems]
Sei $\text{CS} = \langle \text{Conc}, \text{ISA}, \text{ISNOTA}, \text{Feat}, \text{approp}_\text{c} \rangle$ ein Konzeptsystem und demnach $\text{N} = \langle \text{Conc}, \text{ISA}, \text{ISNOTA} \rangle$ ein IS(NOT)A-Netzwerk. Das aus CS abgeleitete Typsystem $\text{TS} = \langle \text{Type}, \sqsubseteq, \text{Feat}, \text{approp} \rangle$ besteht aus folgenden Komponenten:

- Type wird nach Definition 2.6 aus N abgeleitet,
- $\sqsubseteq$ ergibt sich nach Definition 2.7 aus N und
- approp wird entsprechend Definition 2.10 aus N und approp_c konstruiert. $\Diamond$

Carpenter hat bereits gezeigt, daß die angegebenen Konstruktionen der Typen und der Subsumptionsrelation wohldefinierte Ergebnisse liefern. Es bliebe zu zeigen, daß die Konstruktion der Feature-Typisierung wohldefiniert ist und sich damit aus Definition 2.12 ein wohldefiniertes Typsystem ergibt. Da die Konstruktion analog zu [Carpenter 1992] geführt wurde, soll hier auf den Beweis verzichtet werden.

In der Praxis werden häufig Konzeptsysteme gegeben sein, die wie das Beispiel in Abbildung 2.5 nicht alle Kriterien erfüllen. Es ist aber in vielen Fällen möglich, eine spezifizierte ISNOTA-Relation automatisch so zu ergänzen, daß die Bedingung 2.9 (b) erfüllt ist. Dies führt allerdings dazu, daß die Konstruktion des Typsystems inkrementell erfolgen muß. Die in Definition 2.6 gegebene Menge von Typen kann dafür als Menge von Kandidaten herangezogen werden. Jede Inkonsistenz, die sich aus inkompatibler Feature-Typisierung ergibt, führt dazu, daß ein Kandidat gestrichen wird. Dies kann wiederum zu weiteren Inkonsistenzen führen, wenn der gestrichene Typ an anderer Stelle benutzt wurde.

Mit dem vorgestellten Ansatz, Typsysteme aus Konzeptsystemen zu konstruieren, erhält man Mehrfachtypisierung in dem Sinne, daß einer Entität zwar genau ein Typ zugewiesen wird, dieser sich aber aus einer Menge von Konzepten konjunktiv zusammensetzt. Stellt man sich die Konzepte als Rollen vor, so muß ein Objekt, um von einem konjunktiven Typ zu sein, jede der Rollen annehmen können, aus denen der Typ besteht.

Der Begriff Konzept wurde eingeführt, um die Basistypen von den abgeleiteten konjunktiven Typen unterscheiden zu können. In dieser Arbeit werden die Basistypen Konzepte und die abgeleiteten konjunktiven Typen einfach nur Typen genannt. In der Praxis tritt der Fall der Mehrfachtypisierung allerdings erfahrungsgemäß nicht übermäßig häufig auf ([Fowler und Scott 1997]). Im Extremfall, in dem ausschließlich disjunkte Subtypisierung benutzt wird, sind Konzeptsystem und daraus abgeleitetes Typsystem bis auf den im Typsystem explizit definierten Typ $\top$ identisch. Deshalb werden im folgenden oft ein Konzept und der Typ, der aus genau diesem einen Konzept besteht, synonym benutzt, wenn aus dem Kontext klar ist, ob es sich um ein Konzept oder einen Typ handelt.

2.1.4 Eine UML-Notation für Konzeptsysteme

Die Modellierung mit Typen und vor allem die hier verwendete Notation für Konzeptsysteme sei nun anhand eines Beispiel erläutert und motiviert. Mit Rückgriff auf Abschnitt 2.1.1, wo bereits eine ausführliche Einführung in die UML-Notation von Klassen und Typen erfolgt ist, kann die Notation hier relativ knapp beschrieben werden. Das Beispiel wird aus dem Bereich des Verstehens natürlicher Sprache entlehnt, da dies eines der ursprünglichen Einsatzgebiete von Feature Structures darstellt, die wie in der hier vorgestellten Form auf einem Konzeptsystem basieren. Das Beispiel wird zur Veranschaulichung von Feature Structures (Abschnitt 2.3.1) wieder aufgegriffen. Im Anwendungsteil dieser Arbeit (Kapitel 5) wird gezeigt, daß sich Typmodellierung und Feature Structures für viele andere Anwendungsgebiete eignen, durch die Kombination mit Petrinetzen insbesondere für Anwendungen in verteilten Systemen.

Beim Sprachverstehen und -generieren ist ein wichtiges Teilproblem, die Semantik eines natürlichsprachlichen Satzes in eine Datenstruktur abbilden zu können. Hier soll nur ein kleiner Ausschnitt der Thematik eingeführt werden, um das Beispiel zu motivieren. Ausführlicher wird ein ähnliches Beispiel in [Wienberg 1995] vorgestellt, wo weitere Literaturverweise zu finden sind.

Ein Satz besteht im allgemeinen aus verschiedenen Phrasen wie Nominal- und Verbalphrasen. Phrasen besitzen eine sogenannte Bindung, die in Kongruenz zu anderen Phrasen des Satzes stehen muß. Kongruenzen können erst in Abschnitt 2.3.1 mit Hilfe von Feature Structures modelliert werden. Eine Nominalphrase besitzt unter anderem einen Numerus, einen Kasus und einen Genus, während eine Verbalphrase einen Numerus und einen Tempus aufweist.

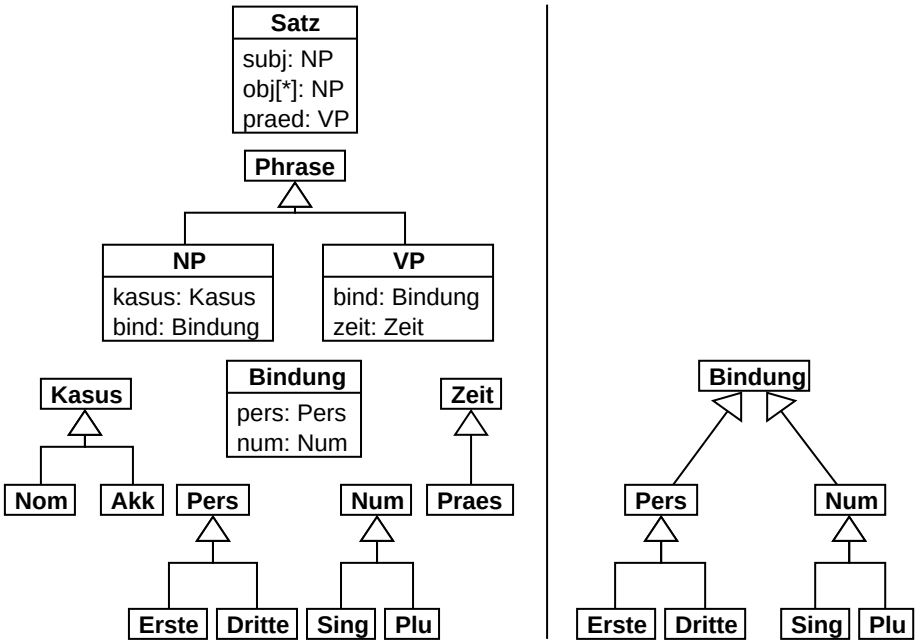

Abbildung 2.6: Ein einfaches Typsystem zur Modellierung der Struktur natürlichsprachlicher Sätze, dargestellt als Konzeptsystem in UML-Notation. Rechts ist eine alternative Modellierung der Bindung mit überlappender Vererbung dargestellt.

Abbildung 2.6 zeigt ein Konzeptsystem bestehend aus Phrase, Nominalphrase (NP), Verbalphrase (VP) und den genannten Feature-Typen in der in Abschnitt 2.1.1 eingeführten Notation für Typen und Attribute.

Der erste Abschnitt (*compartment*) des Typs enthält dessen Namen und implizit das Stereotyp «`type`».

Die ISA-Relation wird durch den UML-Generalisierungspfeil angegeben. Dieses Beispiel zeigt *disjunkte* Vererbung, z.B. zwischen NP und VP (man beachte die in einer Pfeilspitze zusammenlaufenden Generalisierungskanten). Diese Notation bedeutet für Konzeptsysteme, daß die disjunkten Subtypen paarweise in ISNOTA-Relation stehen. Außerdem gelten alle Konzepte der obersten Ebene als disjunkt, d.h. auch diese stehen paarweise in ISNOTA-Relation. Um kompatible Konzepte darzustellen, muß ein expliziter Supertyp angegeben werden, von dem die Konzepte überlappend (*overlapping*, also durch einzelne Pfeilspitzen notierte Generalisierung) erben.

Attribute werden im zweiten Abschnitt angegeben. Wie in UML kann in eckigen Klammern hinter dem Attributnamen eine Multiplizität angegeben werden, wobei in dieser Arbeit für Typdiagramme nur die Multiplizitäten `*` und `1..*` betrachtet werden. Die Typen von mehrwertigen Attributen werden im Konzeptsystem durch typisierte Listenkonzepte dargestellt (siehe Abschnitt 2.3.5).

Abbildung 2.6 enthält auf der rechten Seite eine alternative Modellierung der Bindung. Hier werden statt durch die Attribute pers und num die verschiedenen Werte für Person und Numerus durch überlappende Subtypen modelliert. Durch die überlappende Vererbung sind beliebige Kombinationen aus Person und Numerus möglich. Eine solche Modellierung wird in [Carpenter 1992] vorgestellt, hierbei handelt es sich nach Ansicht des Autoren aber um eine Verwechslung von Vererbung und Assoziation. Ein sinnvolleres Beispiel für Mehrfachvererbung wurde in Abschnitt 2.1.1 in Abbildung 2.2 gezeigt.

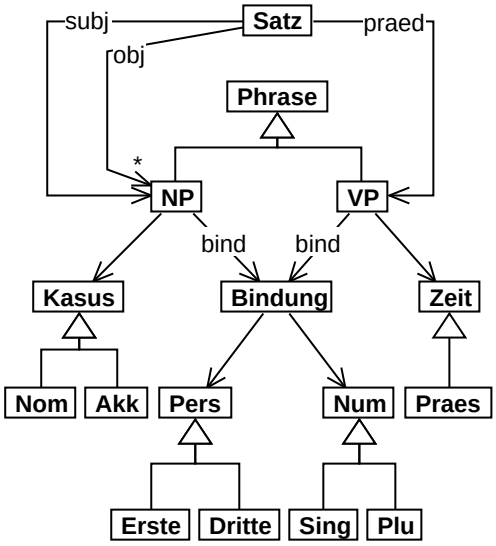

Abbildung 2.7: Das Konzeptsystem aus Abbildung 2.6, wobei Attribute als Assoziationen dargestellt sind.

Wie bei der Diskussion der UML-Notationen in Abschnitt 2.1.1 eingeführt, können Attribute als gerichtete Assoziationen aufgefaßt werden. In Abbildung 2.7 ist das Konzeptsystem aus Abbildung 2.6 dargestellt, wobei Attribute durch Assoziationen notiert werden.

Der Name des Attributs wird als Name der Assoziation übernommen. Wenn für die Assoziation kein Name angegeben wird, heißt das entsprechende Attribut so wie der assoziierte Typ, nur daß aufgrund der Konventionen für Features der Typname klein geschrieben wird. Im Beispiel wurden nach dieser Regel einige Assoziationsnamen weggelassen, beispielsweise die von NP zu Kasus, bei der das entsprechende Feature demnach kasus heißt.

Es sei noch einmal darauf hingewiesen, daß in der hier verwendeten Notation Assoziationen immer * zu 0..1 sind, soweit nicht anders angegeben. Assoziationen können durch die Angabe * am Ziel-Assoziationsende als mehrwertig spezifiziert

werden. Wie mehrwertige Attribute werden mehrwertige Assoziationen auf typisierte Listen abgebildet (siehe Abschnitt 2.3.5).

Wie aufgrund der gegebenen Typmodellierung konkrete Sätze repräsentiert werden, ist Thema der Fortführung dieses Beispiels in Abschnitt 2.3.1.

Nach der Modellierung von Typen wenden wir uns nun der Modellierung eines konkreteren Aspekt der statischen Sicht zu, den Objekten.

2.2 Objekte

In Abschnitt 2.1 wurden Typen als eine Möglichkeit eingeführt, Information zu kategorisieren und diese Kategorien oder Typen in eine Vererbungsrelation zu setzen. Außerdem wurde das Vorhandensein und der Typ von Attributen oder Features an Typen gekoppelt und mit der Vererbungs- oder Subsumptionsrelation in Beziehung gesetzt, was zu den Begriffen des Typsystems und des Konzeptsystems führte.

Dieser Abschnitt wendet sich nun der Betrachtung der eigentlichen Informationsobjekte zu, die ein solches Typsystem nutzen. Datenobjekte können auf vielerlei Art modelliert oder repräsentiert werden: Als Verbunde (*records*), durch mengentheoretische Beschreibungen, als Einträge in (relationalen) Tabellen, als prädikatenlogische Formeln und vieles mehr. Da wir in diesem Kapitel von einer reinen Daten- bzw. Informations- oder Wissensmodellierung ausgehen, wird von *Daten*objekten gesprochen, wenn der Unterschied zur Objektorientierung betont werden soll. Der Begriff der Objekts ist durch die Objektorientierung in der Informatik festgelegt auf eine Kombination aus Datenstruktur und Funktionen oder Verhalten, basierend auf dem Konzept der abstrakten Datentypen. Objekte im Sinne der Objektorientierung sind zwar immer noch nicht eindeutig formal definiert, es besteht aber eine gewisse Einigkeit über Eigenschaften, die ein Objekt erfüllen muß. So muß ein Objekt beispielsweise eine eindeutige Identität besitzen und Operationen kapseln ([Coad und Yourdon 1991, Rumbaugh et al. 1991, Larman 1998]).

Die Rolle der Operationen eines Objekts geht über die von Operatoren abstrakter Datentypen hinaus. Wie in Abschnitt 1.2 eingeführt, kann man in der objektorientierten Analyse funktionales und dynamisches Modell unterscheiden. Die Operationen eines Objekts kapseln demnach nicht nur den funktionalen Zugriff auf dessen Daten, sondern auch den Teil der Dynamik des Systems, an dem dieses Objekt beteiligt ist. Damit vermischen Objekte informations- und prozeßorientierte Modellierung. Wenn dies auch in vielen Fällen, z.B. zur Kapselung von Daten und Verhalten, von Vorteil ist, so sind durch diese Art der Modellierung globale Prozesse nur implizit abbildbar.

Wie in Abschnitt 2.1 bezüglich Typen und Typsystemen wird hier versucht, eine abstrakte Form der Datenmodellierung bereitzustellen, die mächtig genug ist, um typische Phänomene darzustellen, aber gleichzeitig eine anschauliche und verständliche Repräsentation auch komplexer Datenobjekte erlaubt.

Minimalanforderung ist eine Art von Informations- oder Datenobjekt, das Attribute oder Features zuläßt, die als Wert wiederum Datenobjekte zuweisen. Um einen einheitlichen Ansatz zu erreichen, sollten die Daten- und die Wertobjekte tatsächlich gleichartig sein. Diese Auffassung eines Datenobjekts entspricht der aus der Künstlichen Intelligenz stammenden Feature Logik, die sogenannte *Feature Structures* zur Darstellung von Information und Wissen nutzt ([Smolka 1988, Aït-Kaci und Nasr 1986, Aït-Kaci et al. 1992, Carpenter 1990, Carpenter 1992, Pollard und Sag 1994]). Abschnitt 2.3 enthält eine ausführliche Einführung in Feature Structures, gibt Beispiele und definiert die im weiteren Verlauf dieser Arbeit benötigten Begriffe formal. Doch zunächst wird wie bei Typen das Meta-Modell und die Notation von Objekten in UML betrachtet, um den Feature-Structure-Ansatz mit der Objektorientierung in engen Bezug setzen zu können.

Wir beginnen wie für Klassen und Typen mit einem Überblick über Konzepte und Notationen für Objekte in Abschnitt 2.2.1. Die in UML definierten Objektsichten zeigen als eine Art abstrakte Objekte Ähnlichkeiten mit den hier verwendeten Feature Structures und werden deswegen in Abschnitt 2.2.2 vorgestellt. Sodann folgen spezielle UML-Techniken zur Modellierung verteilter Systeme in Abschnitt 2.2.3.

2.2.1 UML-Konzepte und Notationen für Objekte

Neben der Darstellung von Klassen stellt UML eine Notation für Objekte, also Exemplare von Klassen bereit. Verwendet wird diese Notation vor allem in Interaktionsdiagrammen, also Sequenz- oder Kollaborationsdiagrammen, aber auch in einer Variante des Klassendiagramms, dem Objektdiagramm (siehe dazu [Hitz und Kappel 1999] und [Baumgarten 1990]). Das Vorgehensmodell Catalysis ([D'Souza und Wills 1998]) verwendet Objektdiagramme als sogenannte Schnappschüsse (*snapshots*) intensiv. Ein solcher Schnappschuß des Systemzustands zeigt Objekte und deren Attributbelegungen und Assoziationen. Wie bereits diskutiert (Abschnitt 2.1.1) ist es in vielen Fällen die Entscheidung des Modellierers, ob eine Referenz eines Objekts auf ein anderes als Attribut oder als Assoziation bzw. Aggregation oder Komposition dargestellt wird. Meist werden Exemplarvariablen von Basistypen als Attribute, solche von Objekttypen dagegen als Assoziationen, Aggregationen oder Kompositionen dargestellt. Ein Schnappschuß im Sinne von Catalysis resultiert damit in einem sogenannten Objektgraphen, dessen Knoten Objekte (Klassenexemplare) und dessen Kanten Objektbeziehungen sind.

Die UML-Notation eines Objektgraphen sieht folgendermaßen aus. Objekte werden wie Klassen durch einen Kasten dargestellt, der verschiedene Abschnitte (*compartments*, siehe Abschnitt 2.1.1)) enthalten kann. Der erste Abschnitt bezeichnet Namen und Klasse des Objekts. Durch Unterstreichen des Klassennamens, dem ein Doppelpunkt vorangestellt wird, kann ein Objekt von einer Klasse unterschieden werden. Dem Doppelpunkt kann ein optionaler Objektname vorangestellt werden, der ebenfalls unterstrichen wird. Die Attribute werden wie bei Klassen im zweiten

Abschnitt dargestellt, jedoch handelt es sich nun um Attribut*belegungen*. Die Syntax sieht hier vor, daß dem Attributnamen ein Gleichheitszeichen und der Wert folgen. Der Operationsabschnitt wird in Objektdiagrammen üblicherweise weggelassen, da er für alle Objekte einer Klasse gleich ist und somit nur redundante Information enthält. Abbildung 2.8 aus [Hitz und Kappel 1999], S. 25, zeigt Beispiele für die UML-Notation von Objekten.

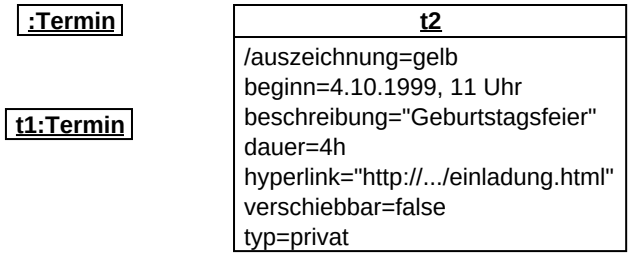

Abbildung 2.8: Darstellung von Objekten in UML-Notation.

In Objektdiagrammen gibt es wie in Klassendiagrammen Assoziationen, nur sind diese hier einfacher strukturiert: Es werden die tatsächlich vorhandenen Beziehungen beschrieben, nicht eine Abstraktion aller zwischen beliebigen Exemplaren der beteiligten Klassen möglichen Beziehungen. Solche Objektbeziehungen können als direkte Darstellung von Referenzen zwischen Objekten interpretiert werden. Die Unterscheidung zwischen *kann*- und *muß*-Beziehungen und verschiedene Multiplizitäten entfallen demnach bei Beziehungen zwischen konkreten Objekten. Navigierbarkeit hat dagegen auch hier eine Bedeutung, da sie die Richtung der Referenz angibt. Der Name eines Assoziationsendes entspricht dem Namen des Attributs bzw. der Exemplarvariablen, welche die zugehörige Referenz speichert.

Ein Beispiel für ein UML-Objektdiagramm

Abbildung 2.9 zeigt ein Beispiel für die UML-Notation eines Objektgraphen mit Objektbeziehungen, die in diesem Fall als Kompositionen modelliert wurden. Das Beispiel benutzt die in vorherigen Abschnitt in Abbildung 2.4 dargestellten Klassen des Java-AWT. Eine konkrete Benutzungsschnittstelle wird in Java durch entsprechenden Code erzeugt, der die entsprechenden AWT-Objekte instantiiert und miteinander in Beziehung setzt. Solcher Code kann per Hand oder mit Hilfe entsprechender Werkzeuge automatisch erstellt werden. Imperativer Java-Code drückt aber nur unzureichend aus, was eigentlich erreicht werden soll: Es soll ein Objektgraph aufgebaut werden, der vom AWT durch entsprechende GUI-Elemente dargestellt wird.

Abbildung 2.9 zeigt einen solchen Objektgraphen und die zugehörige GUI, das

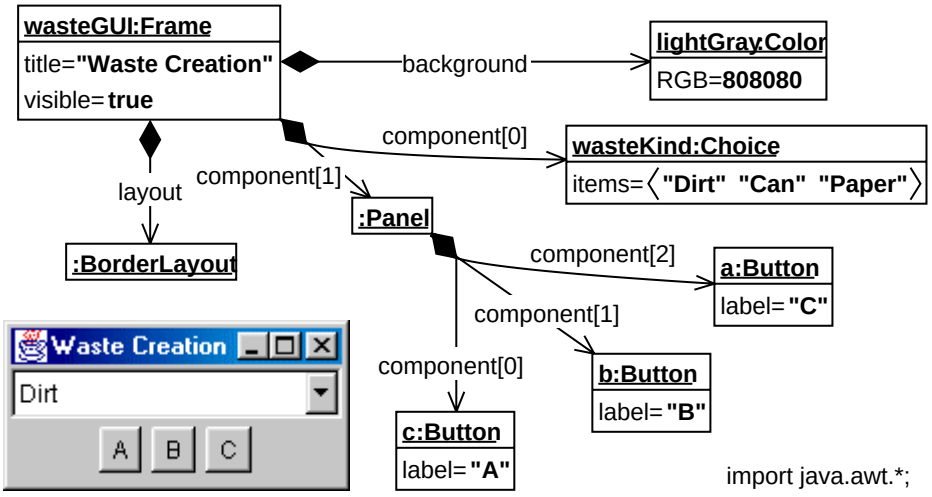

Abbildung 2.9: Darstellung von Objektbeziehungen in UML-Notation.

sich inhaltlich auf das Beispiel in Abschnitt 5.3.4 und technisch auf das Klassendiagramm in Abbildung 2.4 bezieht. Einige der Objekte sind benannt (Bezeichner vor dem Doppelpunkt), einige anonym. Attribute mit Werten von Basistypen oder `Strings` sind direkt im entsprechenden Abschnitt dargestellt, wogegen objektwertige Attribute als Assoziationen modelliert wurden. Mehrwertige Assoziationen wie die von `java.awt.Container` zu `java.awt.Component` werden hier durch einzelne Objektbeziehungen mit einem Index in eckigen Klammern, angelehnt an die Array-Schreibweise, dargestellt. Mehrwertige Attribute werden als Listen von Werten in spitzen Klammern angegeben.

Aggregierte Objekte können in UML alternativ auch direkt im Attributabschnitt dargestellt werden. Hier ist es in UML üblich, die aggregierten Objekte nach der Syntax *Rollenname:Klassenname* zu beschriften, wobei der Rollenname dem Namen der Aggregationsbeziehung bzw. des Aggregationsendes entspricht. Die aggregierten Objekte können dann direkt innerhalb des Attributabschnitts des Aggregats dargestellt werden. Abbildung 2.10 zeigt den Objektgraphen aus Abbildung 2.9 als Aggregat in dieser UML-Notation.

Durch diese Darstellung werden allerdings zwei Dinge vermischt: Einerseits der Name des aggregierten Objekts, andererseits der Name der Aggregation bzw. Rolle. Während ein Aggregationsrollenname in einem Objektdiagramm durchaus mehrfach auftreten kann, sollten Objektnamen zumindest innerhalb eines Diagramms eindeutig gewählt werden, um nicht mit demselben Namen verschiedene Objekte zu bezeichnen. Um hier unterschiedliche Bezeichner für Objekt und Rolle zu erlauben und zusätzlich die Ähnlichkeit von Aggregationsbeziehungen zu (objektwertigen) Attributen zu verdeutlichen, wird in dieser Arbeit eine leicht modifizierte Notation

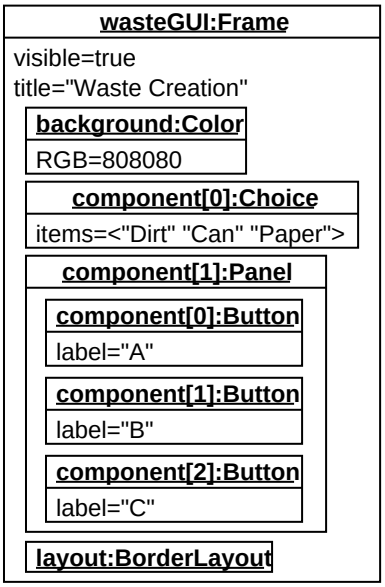

Abbildung 2.10: Alternative Darstellung von Aggregaten in UML-Notation.

verwendet. Der Name des Aggregationsendes wird als Attributname wie bei Attributen von Basistypen gefolgt von einem Gleichheitszeichen notiert, nur daß diesem nicht ein Wert, sondern das aggregierte Objekt folgt, das dadurch einen individuellen Namen zugewiesen bekommen kann. Weiterhin schreibt UML keine eindeutige Notation für mehrwertige Aggregate vor, die hier analog zur Darstellung mehrwertiger Attribute von Basistypen durch Listen in spitzen Klammern notiert werden. Hier wird der Typ der Listenelemente (der *Basistyp* der Liste, siehe Abschnitt 2.3.5) links oben in den spitzen Klammern als Objekttyp notiert. Abbildung 2.11 zeigt den Objektgraphen aus Abbildung 2.9 als Aggregat in der hier verwendeten Variante der UML-Notation.

2.2.2 Objektsichten in UML

Im vorherigen Abschnitt wurde die Unterscheidung zwischen Klassen und ihren Ausprägungen oder auch Exemplaren, den eigentlichen Objekten, dargestellt. In UML werden Objekte vor allem in sogenannten *Kollaborationen* eingesetzt, auf die in Abschnitt 3.1.1 eingegangen wird, da diese zu den verhaltensorientierten UML-Techniken gehören. Für den Moment soll uns nur die Unterscheidung von verschiedenen Abstraktionsebenen in Kollaboration interessieren. Je nachdem, ob Kollaborationen auf *Exemplarebene* (nach [Hitz und Kappel 1999] *Instanzebene*) oder auf *Spezifikationsebene* angegeben werden, basieren sie auf konkreten Objekten und Objektbeziehungen oder auf *Rollen* ([OMG 2000], [Hitz und Kappel 1999]).

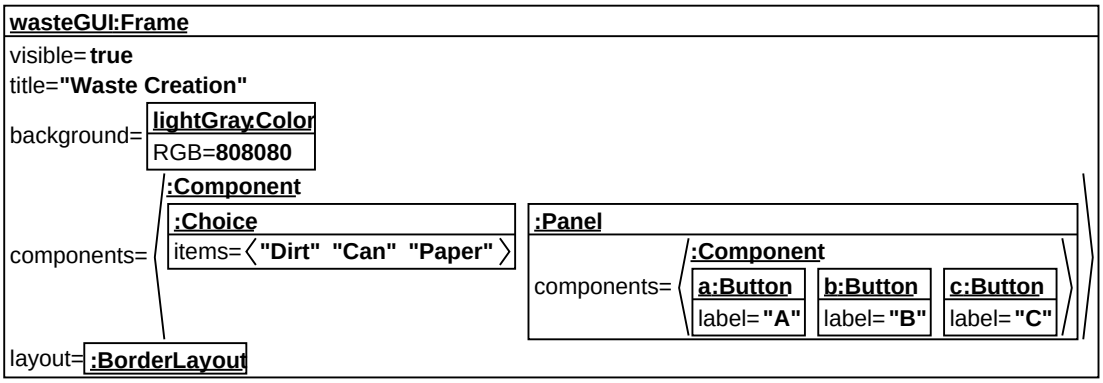

Abbildung 2.11: Darstellung von Aggregaten in der hier verwendeten Variante der UML-Notation.

In UML wird für diese Art Rolle der Begriff *classifier role* verwendet, der in [Hitz und Kappel 1999] in Anlehnung an den inhaltlich verwandten Begriff *view* aus der Datenbankmodellierung als *Sicht* übersetzt wird. Wegen der Verwendung des Begriffs *Sicht* für verschiedene Modellierungssichten sei hier zur Unterscheidung von einer *Objektsicht* die Rede.

Eine Objektsicht definiert demnach die Rolle eines Objekts innerhalb einer Kollaboration. Sie gibt die Eigenschaften an, die ein Objekt aufweisen muß, um in dieser Rolle an der Kollaboration teilnehmen zu können, wobei nicht weiter eingeschränkte Eigenschaften ausgeblendet werden. Zur Realisierung einer Objektsicht kann unter Umständen mehr als ein Objekt benötigt werden ([Hitz und Kappel 1999], S. 127).

Damit stellt eine Objektsicht auf eine Klasse etwas Konkreteres als eine Klasse, jedoch etwas Abstrakteres als ein Objekt dar. Kollaborationen auf Spezifikationsebene, die keinerlei Interaktion spezifizieren, stellen damit *abstrakte Objektdiagramme* dar.

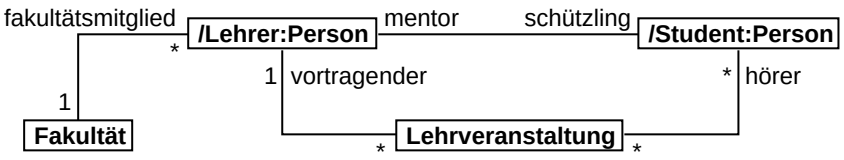

Abbildung 2.12: Eine UML-Kollaboration auf Spezifikationsebene aus [Hitz und Kappel 1999], S. 125.

Abbildung 2.12 zeigt ein solches abstraktes Objektdiagramm aus [Hitz und Kappel 1999], S. 125, wo es als Beispiel für eine Kollaboration auf Spezifikationsebene

angegeben wird. Objektsichten werden ähnlich wie Objekte notiert. Der Name der Objektsicht wird mit einem Schrägstrich eingeleitet. Um eine Verwechslung mit einem Objekt auszuschließen, werden Objektsichten im Gegensatz zu Objekten nicht unterstrichen. Der Name von Objekten in der oben beschriebenen Darstellung kann in Kollaborationen auf Instanzebene durch den Namen der Objektsicht ergänzt werden, wobei der Schrägstrich als Trennzeichen dient.

Die Einführung von Objektsichten ist für diese Arbeit von Bedeutung, da die für FSNets verwendeten Feature Structures Objektsichten sehr ähnlich sind.

2.2.3 Verteilung und verteilte Objekte in UML

UML bietet mit dem *Verteilungsdiagramm* (*deployment diagram*) eine Modellierungstechnik, die speziell auf den Entwurf verteilter Systeme abzielt. Verteilungsdiagramme modellieren die Verteilung von Software-Komponenten auf verschiedenen Rechnerknoten. Dabei ist, ähnlich wie in Klassendiagrammen, keine Modellierung von Dynamik möglich, wohl aber eine Darstellung *potentieller* Dynamik durch Abhängigkeiten (siehe unten).

Der Begriff der *Komponente* ist in UML stark von der Implementierung geprägt. Nach [Hitz und Kappel 1999], S. 164, repräsentiert eine Komponente „einen physischen, austauschbaren Teil des implementierten Systems, der eine Menge von Schnittstellen anbietet und auch deren Realisierung zur Verfügung stellt.“ Diese Definition macht deutlich, daß es sich bei Komponenten in UML um äußerst konkrete Teile des Softwaresystems handelt. Da Komponenten „physisch“ verstanden werden, enthalten sie stets Klassen und Schnittstellen des Implementierungsmodells. UML unterscheidet weiterhin Quellcode-, Binärcode- und ausführbare Komponenten. Bei ausführbaren Komponenten werden Komponententyp und Komponentenexemplar[1] unterschieden, wobei in UML „Komponente“ und „Komponententyp“ synonym gebraucht werden. Während Komponenten Klassen und Schnittstellen enthalten, setzen sich Komponentenexemplare aus Klassenexemplaren, also Objekten (und Klassenobjekten, falls diese betrachtet werden) zusammen.

Eine Komponente wird in UML als ein Kasten mit zwei kleinen Kästen an der linken Seite notiert (siehe Abbildung 2.13). Ein Komponententyp erhält einen Bezeichner, während ein Komponentenexemplar analog zu einem Objekt einen optionalen Namen und Typ angibt, die durch einen Doppelpunkt getrennt und unterstrichen dargestellt werden.

Im *Verteilungsdiagramm* wird spezifiziert, welche Komponenten des Softwaresystems zur Laufzeit auf welchen Rechnerknoten liegen. Obwohl UML Verteilungsdiagramme auf Typ- und Exemplarebene erlaubt, scheint nach [Hitz und Kappel 1999] nur eine Anwendung auf Exemplarebene sinnvoll. Als Motivation für ein Verteilungsdiagramm mit Komponenten*typen* wäre allerdings ein Applikations-Server denkbar,

[1] In [Hitz und Kappel 1999] wird statt Komponentenexemplar der Begriff Komponenteninstanz verwendet.

der zur Laufzeit Komponententypen zu Komponentenexamplaren instantiiert.

Rechnerknoten in Verteilungsdiagrammen werden als „dreidimensionale“ Kästen gezeichnet und enthalten Komponenten. Analog zu Komponenten wird zwischen *Rechnerknotentypen* und *Rechnerknotenexemplaren* unterschieden. Ebenso erhalten Rechnerknoten(-exemplare) einen Namen. Rechnerknoten können durch Kanten verbunden werden, die mögliche Kommunikationsverbindungen spezifizieren, und mit der Art des verwendeten Netzwerks stereotypisiert werden können.

Da Verteilungsdiagramme auf Komponenten aufbauen, sind auch diese an ein konkretes Implementierungsmodell gebunden. Obwohl sich diese Arbeit mit der Modellierung von Verteilung auf einer wesentlich abstrakteren Ebene befaßt, wollen wir das Verteilungsdiagramm anhand eines Beispiels näher betrachten, da sich diese Technik möglicherweise auch auf eine abstraktere Ebene übertragen läßt.

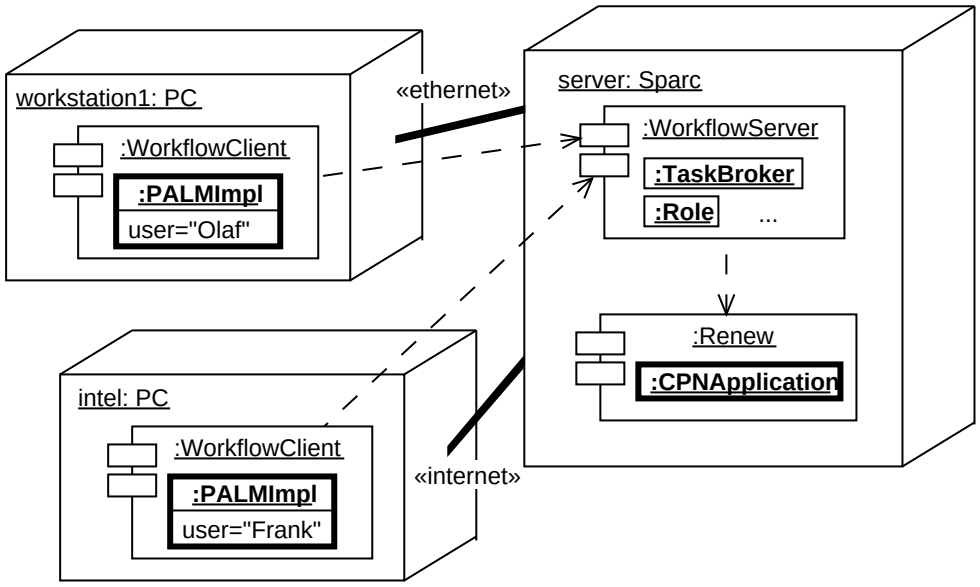

Abbildung 2.13: Ein UML-Verteilungsdiagramm, das die grobe Architektur der Renew-Workflow-Erweiterungen modelliert.

Abbildung 2.13 zeigt ein Verteilungsdiagramm mit drei Rechnerknoten, die verschiedene Komponenten enthalten. Es handelt sich dabei um eine (sehr grobe) Darstellung der Verteilungsarchitektur der Renew-Workflow-Erweiterungen, die in Abschnitt 3.5.2 vorgestellt werden. Es handelt sich dabei um eine klassische Client-Server-Architektur. Die WorkflowClient-Komponentenexemplare laufen auf den beiden Klientenrechnern (in Abbildung 2.13 links), während sich die Ausführungsmaschine (Renew) und die Workflow-Erweiterungen (WorkflowServer) auf dem Server-Rechner (rechts) befinden.

Aktive Objekte, also Objekte, die in einem eigenen Prozeß laufen, werden in UML durch ein fett umrandetes Rechteck dargestellt. Da der WorkflowServer von Renew und dem WorkflowClient aufgerufen wird, ist er selbst kein aktives Objekt.

Die Klienten sind mit dem Server durch eine ethernet- bzw. internet-Verbindung verknüpft. Zusätzlich sind die *Abhängigkeiten* der Komponenten dargestellt. Hier könnte auch noch genauer angegeben werden, auf welche Schnittstellen der Komponente die Klienten-Komponente zugreift.

Die bisher gezeigten Möglichkeiten lassen nur eine rein *statische* Beschreibung der Verteilung von Software-Komponenten zu. Auf die dynamischen Modellierungstechniken von UML kommen wir ausführlich in Kapitel 3 zurück. Als informationsorientierte Technik behandeln wir an dieser Stelle die Spezifikation von *potentieller* Dynamik, wie sie auch in Klassendiagrammen auftritt.

In Klassendiagrammen kann über Abhängigkeiten beispielsweise angegeben werden, welche Klasse Objekte welcher anderen Klasse instantiiert. Dies ist ein dynamischer Aspekt, da Objekte erst zur Laufzeit instantiiert werden, und wird im Klassendiagramm deshalb nur abstrakt betrachtet: Es wird nicht beschrieben, *wann*, also unter welchen Bedingungen oder in welcher Aktivität, Objekte instantiiert werden, nur *daß* dies durch eine bestimmte Klasse geschieht.

Ähnlich verhält es sich mit der Möglichkeit, mit UML die *Migration* von Objekten ([Boger 1999]) zu modellieren. Im Unterschied zum oben beschriebenen Fall gibt es aber keine andere Diagrammart, in der das *Wann* genauer beschrieben werden kann. Damit sind die Möglichkeiten der Modellierung der Dynamik verteilter Systeme mit UML sehr begrenzt.

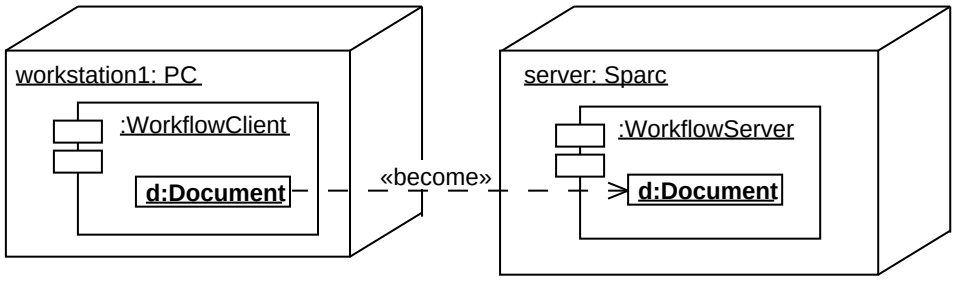

Abbildung 2.14: Modellierung von Objektmigration mit einem Verteilungsdiagramm.

Abbildung 2.14 zeigt ein Beispiel für die (statische) Modellierung einer Objektmigration in UML. Nehmen wir an, ein Dokument d wird innerhalb des Workflows auf der Workflow-Client-Workstation bearbeitet. Das Dokument soll nach der Bearbeitung wieder auf dem Server abgelegt werden, bei der Übertragung aber seine (Objekt-)Identität behalten. Eine solche Objektmigration wird in UML wie in Abbildung 2.14 zu sehen durch die mehrfache Notation des Objekts und eine Abhängigkeit

mit dem Stereotyp «becomes» modelliert. Details wie Migrationszeitpunkt oder Reihenfolge der Migration lassen sich weder mit diesem noch mit anderen Diagrammen angeben.

Ähnliche Stereotypen für Verteilung müßten für viele weitere Diagrammarten von UML vorgegeben werden. Denkbar wäre beispielsweise ein Stereotyp «remote» für Assoziationen in Objekt- oder Interaktionsdiagrammen, die entfernten Referenzen entsprechen.

Obwohl UML eine moderne objektorientierte Modellierungssprache darstellt, zu deren Entwurfszeit die Bedeutung verteilter Systeme bereits hätte absehbar sein sollen, gibt es nur wenige, unzureichende Konzepte und Notationen, welche die Modellierung verteilter System angemessen unterstützen.

Eine wichtige Voraussetzung für die Betrachtung von Verteilung ist eine Darstellung von Objekten und Objektgraphen auf einer hinreichend abstrakten Ebene. In UML kommen die Objektsichten diesem Anspruch am nächsten. Im nächsten Abschnitt werden Feature Structures, die als eine Formalierung von Objektsichten betrachtet werden können, eingeführt und formal definiert.

2.3 Feature Structures: Attribut-Wert-Strukturen als universelle Informations- und Wissensmodelle

In den beiden vorhergehenden Abschnitten wurden Typen und Objekte als die Grundbausteine der informationsorientierten Modellierung eingeführt. In dieser Arbeit wird Information durch eine dritte Kategorie repräsentiert, die sich wie die in Abschnitt 2.2.2 eingeführten Objektsichten in einer Abstraktionshierarchie *zwischen* Typen und Objekten eingliedert. Objektsichten sind konkreter als Typen und Klassen, da sie für Objekte stehen, aber abstrakter als Objekte, da sie für Objektsichten verschiedene konkrete Objekte eingesetzt werden können.

In dieser Arbeit werden *Feature Structures* zur Informationsrepräsentation herangezogen. Eine Feature Structure stellt eine abstrakte Beschreibung eines Datenobjekts durch teilweise Spezifikation seines Typs und seiner Attribute dar. Feature Structures repräsentieren *Information*, die *über* Objekte vorliegt, nicht die Objekte selbst. Diese Information muß nicht so speziell sein, daß sich aus einer Feature Structure ein eindeutiges Objekt ergibt, weshalb diese als Repräsentant einer *Menge von Objekten* angesehen werden kann. Man spricht hier von einer Feature Structure als *unterspezifizierte* Beschreibung.

Feature Structures haben den Vorteil, daß ihr *Informationsgehalt* eindeutig verglichen und unifiziert (vereinigt) werden kann. Dafür dienen Subsumption und Unifikation.

Man kann eine Feature Structure auf zwei Weisen betrachten: Erstens als die Beschreibung *eines komplexen* Datenobjekts, oder zweitens als die Beschreibung vieler einzelner Datenobjekte. In dieser Arbeit werden je nach Bezug beide Positionen ein-

genommen. Die erste Betrachtungsweise liegt näher an klassischen Datenstrukturen wie verschachtelten Verbunden (*records*), während die zweite dem objektorientierten Begriff des Objektgraphen entspricht, in dem jeder Knoten als einzelnes Objekt angesehen wird. Mit der in Abschnitt 2.1.1 eingeführten Abbildung von Assoziationen auf Attribute und Aggregation kann auch in der Objektorientierung die erste Betrachtungsweise eingenommen werden.

Es folgt in Abschnitt 2.3.1 eine allgemeine Einführung in Feature Structures, die anhand eines Beispiels deren Anwendung veranschaulicht und motiviert. Die weiteren Unterabschnitte zeigen die formale Definition von Feature Structures. Abschnitt 2.3.2 definiert grundlegende Begriffe für Feature Structures. Abschnitt 2.3.3 widmet sich der *Subsumption* und *Unifikation*, die nun vom unterliegenden Typsystem auf die Ebene der Datenobjekte übertragen wird. Da Feature Structures als Graphen definiert werden, spricht man hier auch von *Graphunifikation*. Als eigener Beitrag wird der Feature-Structure-Formalism um den *Vorpfad-Operator* erweitert, der sich bei der Manipulation von Objektgraphen als nützlich erweist und zur Definition von FSNets in Abschnitt 4 eingesetzt wird. Abschnitt 2.3.4 behandelt typisierte Feature Structures, welche die in Abschnitt 2.1.2 definierten Feature-Typisierungen beachten. Schließlich führt Abschnitt 2.3.5 einige hilfreiche abkürzende Notationen anhand von Beispielen ein und definiert Tupel und Listen als Standard-Datentypen.

2.3.1 Einführung in Feature Structures

Feature Structures stammen aus der KI, speziell der Wissenrepräsentation, und werden dort vor allem im Bereich des Sprachverstehens und der Sprachgenerierung eingesetzt. Da Feature Structures durch Prädikatenlogik mit Funktionen beschrieben werden können, spricht man auch von *Feature Logik* (*feature logic*). In der Literatur sind einige wenige Anwendungen von Feature Structures außerhalb der KI zu finden, wie beispielsweise die Unterstützung von Konfigurationsmanagement ([Zeller 1994]) und Versionsverwaltung ([Zeller und Snelting 1996], [Zeller und Snelting 1997]). Es gibt aber andere Ansätze, die logische Beschreibungen (*constraints*) und Objektorientierung in Verbindung bringen, wie OCL ([Warmer und Kleppe 1998]) und die Constraint-Diagramme in [Kent 1997].

Die Entwicklung der Feature Structures ist ähnlich verlaufen wie die von Datenstrukturen zu Datenobjekten. Eine Feature Structure ist ein gerichteter, wurzelbehafteter Graph. In frühen Ansätzen wie [Kasper und Rounds 1986], [Vijay-Shanker und Joshi 1988] und [Pollard und Sag 1987] enthielten nur Blattknoten atomare Information und man beschränkte sich auf azyklische Strukturen, wodurch diese einfachen Feature Structures nicht viel mehr als verschachtelte Verbunde mit geteilten Strukturen (*structure sharing*) darstellen. In [Moshier 1988] und [Smolka 1988] werden zyklische Feature Structures erlaubt und deren Unifikation wird betrachtet. Durch die Aufarbeitung der Feature-Structure-Theorie und die Erweiterung um ein Typsystem mit Mehrfachvererbung durch Carpenter in

[Carpenter 1992] und [Pollard und Sag 1994] fand eine Vereinheitlichung der Konzepte statt. In Feature Structures nach Carpenter gibt es nur noch eine Art von Knoten, die einen Typ aus der Typhierarchie zugeordnet bekommen. Atome, die in früheren Ansätzen als eine spezielle Art von Knoten definiert wurden, können durch Knoten mit speziellen Typen (extensionale Typen ohne Subtypen) dargestellt werden. [Carpenter 1992], S 33 ff., stellt weitere Ansätze und Varianten von Feature Structures mit detaillierten Literaturangaben dar.

Das in Abschnitt 2.1.4 begonnene Beispiel der Modellierung natürlichsprachlicher Sätze soll nun zur allgemeinen Einführung in Feature Structures fortgeführt werden.

Basierend auf dem in Abschnitt 2.1.4 in Abbildung 2.7 vorgestellten Konzept- bzw. dem daraus abgeleiteten Typsystem kann man die Semantik eines einfachen Satzes aus Subjekt, Prädikat und Objekt als Feature Structure wie in Abbildung 2.15 darstellen. Hier soll zunächst die auf der rechten Seite der Abbildung gezeigte *Graphnotation* beschrieben werden, die eine Feature Structure als einen *gerichteten, wurzelbehafteten Graphen* notiert. Die Knoten werden mit den ihnen zugeordneten *Typen* aus dem Typsystem beschriftet, die Kanten mit *Features*. Eine Kante ohne Ausgangsknoten zeigt den *Wurzelknoten* der Feature Structure an.

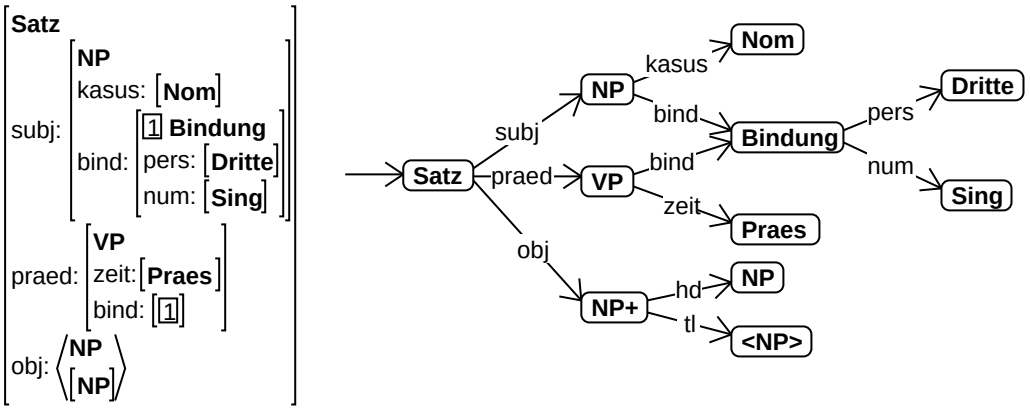

Abbildung 2.15: Eine Feature Structure über dem Typsystem aus Abbildung 2.7, die einen Satz darstellt, als AVM und als Graph.

Die abgebildete Feature Structure stellt einen Satz dar (der Wurzelknoten trägt den Typ Satz), der aus Nominalphrase, Verbalphrase und einer weiteren Nominalphrase besteht, die durch die Features subj, praed und obj zugewiesen werden. Wenn wie hier zwei Werte denselben Typ tragen (NP), zeigt sich der Vorteil benannter Attribute. In einem prädikatenlogischen Term der Art `Satz(NP,VP,NP)` wäre ohne weitere Vereinbarungen nicht deutlich, welche Rolle die drei Argumente spielen. Weiterhin hängt die Bedeutung von der Reihenfolge ab, während diese in Feature Structures unerheblich ist.

Die Knoten unter subj und praed enthalten wiederum weitere Features. Ein interessantes Phänomen tritt dort auf, wo ein Knoten zwei Eingangskanten aufweist: Die Bindung (der Knoten mit dem Typ Bindung) wird sowohl von der Subjekt-NP als auch von der Prädikat-VP referenziert. Dies repräsentiert die Strukturgleichheit oder auch *Koreferenz* zwischen Phrasen, die Feature Structures deshalb so elegant ausdrücken können, weil sie als Graphen (im Gegensatz zu Bäumen) Koreferenzen erlauben und so den Unterschied zwischen Gleichheit und Identität darstellen können. Eine Motivation, Feature Structures als Datenstruktur für Sätze zu nutzen, ist, daß in der Semantik natürlicher Sprache häufig Phänomene wie Strukturgleichheit auftreten. Wenn Knoten in Feature Structures auch nicht explizit benannt sind (die Beschriftungen sind die Typen der Knoten, nicht deren Namen), so besitzen sie doch eine eindeutige Identität, so daß sie referenziert werden können.

Die Darstellung von Feature Structures als Graphen ist zwar sehr anschaulich, kann aber für größere Strukturen mit vielen Koreferenzen auch unübersichtlich werden und erfordert spezielle Werkzeugunterstützung, um sie erstellen zu können. Deshalb wurde schon früh eine matrizenartige Darstellung eingeführt, die Attribut-Wert-Matrix (*attribute-value-matrix,* AVM, [Carpenter 1992]) genannt wird und sich an der Auffassung einer Feature Structure als verschachtelter Verbund (*record*) orientiert. Eckige Klammern umschließen in dieser Darstellung einen Knoten mitsamt seinem Typ und seinen Features, sowie allen von diesen aus erreichbaren Substrukturen. Abbildung 2.15 zeigt auf der linken Seite die bereits beschriebene Feature Structure als AVM.

Der Typ des jeweiligen Knotens wird als erste Zeile innerhalb der eckigen Klammern angegeben. Für jedes Feature wird das Feature selbst, ein Doppelpunkt und der diesem Feature zugeordnete Wert angegeben, der rekursiv wieder als AVM dargestellt wird. Strukturgleichheiten, also Abweichungen von der Baumstruktur, werden durch sogenannte *Knotenreferenzen* (*tags*) darstellt. Wenn aus dem Kontext deutlich wird, daß Referenzen auf Feature-Structure-Knoten gemeint sind, wird einfach nur von Referenzen gesprochen. Referenzen benennen Knoten durch eine in einem Kasten dargestellte natürliche Zahl, welche direkt dem Typ des Knotens vorangestellt wird. Als Erweiterung gegenüber [Carpenter 1992] erlauben wir auch alphanumerische Bezeichner für Knotenreferenzen. Zwei Knoten, welche innerhalb einer AVM denselben Bezeichner als Knotenreferenz tragen, notieren *denselben* Knoten. Üblicherweise wird der Knoten nur bei einem seiner Auftreten ausführlich beschrieben. Jedes weitere Auftreten eines bereits ausführlich beschriebenen Knotens als Feature-Wert wird auf die entsprechende Knotenreferenz in eckigen Klammern reduziert.

Die Verwendung von Knotenreferenzen ähnelt der von Variablen in anderen Formalismen, vor allem, wenn Feature Structures Regeln darstellen. Man könnte deshalb überlegen, Knotenreferenzen immer statt durch Zahlen durch symbolische Bezeichner zu repräsentieren. Da Werte in Feature Structures aber unter einem benannten und typisierten Feature auftreten, ergibt sich die Bedeutung meist aus diesem Kontext, weshalb in dieser Arbeit meist Zahlen als Knotenreferenzen be-

nutzt werden. Eine wohlüberlegte Wahl von Feature-Namen ist in diesem Fall für eine aussagekräftige Modellierung mit Feature Structures besonders wichtig.

Graph- und AVM-Notation weisen die genannten Vor- und Nachteile auf. Da es im Einzelfall manchmal schwierig sein kann, diese Vor- und Nachteile abzuwägen und sich für eine Darstellung zu entscheiden, wird in dieser Arbeit gelegentlich eine neue Notation benutzt, die eine Mischform der Graph- und AVM-Notation darstellt. In dieser Notation sei es erlaubt, in einem Knoten statt des Typs eine Feature Structure in AVM-Notation darzustellen. Alle Feature-Kanten, die auf einen solchen komplexen Knoten zeigen oder von diesem ausgehen, beziehen sich auf den Wurzelknoten der als AVM dargestellten Substruktur. Koreferenzen auf Substrukturen eines komplexen Knotens von außen sind in dieser Schreibweise nicht ausdrückbar.

Wichtig ist, daß Knotenreferenzen innerhalb der AVM eines komplexen Knotens nur lokal gültig sind. Knotenreferenz $\boxed{1}$ bezeichnet demnach in zwei AVMs, die komplexe Knoten derselben Feature Structure beschriften, *nicht* denselben Knoten. Dies stimmt mit der Konvention überein, daß auch innerhalb eines Ausdrucks, der sich aus mehreren Feature Structures zusammensetzt, Knotenreferenzen nur lokale Gültigkeit aufweisen (siehe oben). Die Mischnotation wird insbesondere bei der Darstellung von Erreichbarkeitsgraphen und Prozessen in Abschnitt 4.2 genutzt.

In Abschnitt 2.1.1 wurden verschiedene UML-Notationen für Objektgraphen vorgestellt. Es wurde dargestellt, daß Attribute direkt innerhalb der Klasse bzw. des Typs oder als Assoziation dargestellt werden können. Die Mischnotation von AVM- und Graphnotation erlaubt dem Modellierer in diesem Sinne, die jeweils treffendste Notation frei zu wählen.

Nach den nächsten drei Abschnitten, die Feature Structures formal betrachten, werden in Abschnitt 2.3.5 noch weitere Notationsmöglichkeiten definiert.

2.3.2 Formale Definition von Feature Structures

In dieser Arbeit werden Feature Structures nicht in ihrem ursprünglichen Anwendungsgebiet der KI eingesetzt, sondern zur abstrakten Beschreibung von Objektgraphen verwendet. Dieser Abschnitt stellt viele der formalen Definitionen zu Feature Structures aus [Carpenter 1992], aber auch etliche hier eingeführte Erweiterungen vor.

FSNets werden in Kapitel 4 so definiert, daß Feature Structures als Objekt- und Transitionsregelbeschreibungen eingesetzt werden. Für die formale Definition der FSNets sind neue Definitionen für Operatoren erforderlich, welche der gezielten Trennung und Unifikation von Substrukturen in Feature Structures dienen: Der Vorpfad-Operator und der Pfadgleichungskonstruktor. Weiterhin wird hier eine modifizierte Typinferenzprozedur definiert, da in der Objektorientierung im Gegensatz zur KI üblicherweise keine globale Typinferenz wie bei Carpenter verwendet wird.

Es werden Sätze über die Eigenschaften der neuen Operatoren und Funktionen aufgestellt und bewiesen. Damit sind die Voraussetzungen geschaffen, um Eigen-

schaften der FSNets, vor allem in bezug auf Wert- und Referenzmarkierungen (siehe Abschnitt 4.2.3 formal zeigen zu können.

Folgende Definition einer Feature Structure entspricht bis auf minimale Abweichungen in der Notation[2] Carpenters Definition in [Carpenter 1992] und liegt nahe an der Darstellung einer Feature Structure als Graph, wie sie im vorherigen Abschnitt vorgestellt wurde. Carpenter gibt alternative, äquivalente Definitionen: eine logische Axiomatisierung und die *abstrakte Feature Structure*, die eine Feature Structure durch die von ihr definierten Pfade und Pfadgleichungen beschreibt. Auf diese Alternativen wird hier nicht weiter eingegangen, weil die Darstellung als Graph am besten zu den im vorherigen Abschnitt vorgestellten Konzepten aus der Objektorientierung paßt.

Definition 2.13 [Feature Structure]
Eine Feature Structure über einem Typsystem $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ sei ein Tupel $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle$, wobei:

(a) Q die endliche Menge der Knoten,

(b) $\bar{q} \in \mathrm{Q}$ der Wurzelknoten,

(c) $\theta : \mathrm{Q} \rightarrow \mathrm{Type}$ die totale Knotentypisierungsfunktion und

(d) $\delta : \mathrm{Q} \times \mathrm{Feat} \rightarrow \mathrm{Q}$ die partielle Feature-Knoten-Funktion ist. ◊

Sei $\mathrm{FS}_{\mathrm{TS}}$ die Menge aller Feature Structures über dem Typsystem TS. Es wäre auch möglich, das Typsystem als Komponente in die Feature Structure mit aufzunehmen. Da in dieser Arbeit aber meist Feature Structures über einem gemeinsamen Typsystem betrachtet werden, wurde die oben definierte Darstellung gewählt.

Obwohl sich kaum semantische Gemeinsamkeiten finden lassen, gleicht eine Feature Structure strukturell einem Moore-Automaten. Als erste hingewiesen haben auf diesen eher zufälligen Umstand Kasper und Rounds in [Kasper und Rounds 1986]. Zum Vergleich sei hier die Definition eines deterministischen, reduzierten Moore-Automaten aus [Conway 1971] wiedergegeben.

Definition 2.14 [Moore-Automat]
Eine endliche Maschine (endlicher Automat, Moore-Automat) ist ein Tupel $\langle \mathrm{S}, \mathrm{I}, \mathrm{O}, \mathrm{t}, \mathrm{o}, \iota \rangle$ aus

(a) einer endlichen Menge S von *Zuständen* (states)

(b) einer endlichen Menge I von *Eingaben* (inputs)

[2] Carpenter definiert die Funktion δ aus Definition 2.13 mit einer umgekehrten Reihenfolge der Argumente. Um die objektorientierte Sichtweise von Feature Structures zu betonen, werden Kanten wie Attribute ihren Ausgangsknoten zugeordnet, weshalb dem Autoren die gewählte Reihenfolge im Zusammenhang dieser Arbeit angemessener erscheint.

(c) einer endlichen Menge O von *Ausgaben* (outputs)

(d) einer *Transitionsfunktion* (transition function) $t : S \times I \rightarrow S$,

(e) einer *Ausgabefunktion* (output function) $o : S \rightarrow O$ und

(f) einem *Anfangszustand* (initial state) $\iota \in S$. ◊

Dabei entsprechen die Zustände S des Automaten der Menge Q der Knoten der Feature Structure, die Eingaben I der Menge Feat und die Ausgaben O der Menge Type (beide aus dem Typsystem der Feature Structure), die Transitionsfunktion t der Feature-Knoten-Funktion δ, die Ausgabefunktion o der Knotentypisierungsfunktion θ und der Anfangszustand ι dem Wurzelknoten $\bar{q}$.

Obwohl Feature Structures normalerweise nicht wie Automaten Sprachen, Protokolle oder Zustandsmodelle beschreiben[3], lassen sich doch formale Notationen aus der Welt der Automaten auf Feature Structures übertragen. Die weiteren Definitionen wie Subsumption und Unifikation haben aber nach Ansicht des Autoren weder eine sinnvolle Entsprechung in der Welt der Automaten, noch werden in dieser Arbeit andere Automaten-Begriffe wie beispielsweise die akzeptierte Sprache auf Feature Structures angewandt.

Die Feature-Knoten-Funktion δ kann man analog zur Kantenschreibweise in Automaten darstellen als $q \xrightarrow[\delta]{f} q' \iff \delta(q, f) = q'$. Wenn deutlich ist, welche Funktion gemeint ist, kann das δ unter dem Pfeil entfallen.

Der Begriff des *Wortes* aus der Automatentheorie wird bei Feature Structures aus der Graphentheorie entlehnt und lautet dementsprechend *Pfad*. Zur Unterscheidung von anderen Pfad-Definitionen sagt man auch *Feature-Pfad*. Während in der Graphentheorie nur solche Folgen von Kantenbezeichnern als Pfad zulässig sind, die im Graphen tatsächlich vorkommen, werden in der folgenden Definition zunächst alle Worte aus Features erlaubt.

Definition 2.15 [Feature-Pfad]
Ein *Feature-Pfad* oder kurz Pfad π über einer Feature-Menge Feat ist ein Folge von Features aus Feat. Wir schreiben Feat^* für die Menge aller Pfade über einer Featuremenge Feat, Feat^+ für die Menge aller nicht-leeren Pfade und ε für den leeren Pfad (Pfad der Länge Null). ◊

Analog zu Definitionen aus der Graphentheorie ist es sinnvoll, die Feature-Knoten-Funktion δ auf Pfade zu erweitern. Hier findet die Unterscheidung statt, ob ein Pfad tatsächlich im Graphen der Feature Structure vorhanden ist.

Definition 2.16 [Pfad-Knoten-Funktion]
Für eine Feature Structure $F = \langle Q, \bar{q}, \theta, \delta \rangle$ über einem Typsystem $TS = \langle \text{Type}, \sqsubseteq, \text{Feat}, \text{approp} \rangle$ werde die Feature-Knoten-Funktion δ folgendermaßen auf Pfade erweitert:

[3]Feature Structures beschreiben Information jeglicher Art. Wie anhand des Beispiels in Abschnitt 5.3.4 gezeigt wird, können Feature Structures auch als deklarative Repräsentation von Verhalten genutzt werden.

(a) $\delta(q, \varepsilon) := q$

(b) $\forall f \in \text{Feat}, \pi \in \text{Feat}^* : \delta(q, f\pi) := \delta(\delta(q, f), \pi)$ $\Diamond$

Auch für die Knotentypisierungsfunktion gibt es eine kanonische Erweiterung, und zwar von Knoten auf Feature Structures. Man beachte, daß diese erweiterte Funktion θ damit nicht mehr als Teil einer Feature Structure, sondern „global“ definiert ist. Durch den Argumenttyp ist stets unterscheidbar, welche Funktion gemeint ist.

Definition 2.17 [Feature-Structure-Typ-Funktion]
Der Typ einer Feature Structure $F = \langle \text{Q}, \bar{q}, \theta, \delta \rangle \in \text{FS}_{\text{TS}}$, notiert als $\theta(F)$, sei definiert als der Typ des Wurzelknotens von F, also $\theta(F) := \theta(\bar{q})$. $\Diamond$

Wir sprechen von einer zusammenhängenden Feature Structure, wenn diese einen zusammenhängenden Graphen darstellt (siehe [Conway 1971], *truncation by inaccessibility*).

Definition 2.18 [zusammenhängende Feature Structure]
Eine Feature Structure $F = \langle \text{Q}, \bar{q}, \theta, \delta \rangle$ über einem Typsystem $\text{TS} = \langle \text{Type}, \sqsubseteq, \text{Feat}, \text{approp} \rangle$ heißt *zusammenhängend* (connected), wenn die Menge der Knoten Q vom Wurzelknoten $\bar{q}$ aus erreichbar ist, also:

$$\text{Q} = \{\delta(\bar{q}, \pi) \mid \pi \in \text{Feat}^*\}$$ $\Diamond$

Sei CFS_{TS} die Klasse der zusammenhängenden Feature Structures (*connected feature structures*) über dem Typsystem TS.

Mit der Pfad-Knoten-Funktion δ greift man auf einzelne Knoten der Feature Structure zu. Eine Substruktur einer Feature Structure kann wiederum als Feature Structure dargestellt werden, was zum Begriff des *Pfadwerts* ([Carpenter 1992]) führt.

Definition 2.19 [Pfadwert / @-Operator]
Sei $\text{TS} = \langle \text{Type}, \sqsubseteq, \text{Feat}, \text{approp} \rangle$ ein Typsystem. Der *Pfadwert* (path value) einer Feature Structure unter einem Pfad sei definiert durch den *Pfadwert-Operator* (@) : $\text{FS}_{\text{TS}} \times \text{Feat}^* \hookrightarrow \text{FS}_{\text{TS}}$ mit $\forall F = \langle \text{Q}, \bar{q}, \theta, \delta \rangle \in \text{FS}_{\text{TS}}, \pi \in \text{Feat}^*$:

$$F@\pi := (@)(F, \pi) := \begin{cases} \langle \text{Q}', \bar{q}', \theta', \delta' \rangle \in \text{FS}_{\text{TS}} & \text{falls } \delta(\bar{q}, \pi) \text{ definiert} \\ \text{undefiniert} & \text{sonst,} \end{cases}$$

wobei

(a) $\bar{q}' := \delta(\bar{q}, \pi)$,

(b) $\text{Q}' := \{\delta(\bar{q}', \pi') \mid \pi' \in \text{Feat}^*\} = \{\delta(\bar{q}, \pi\,\pi') \mid \pi' \in \text{Feat}^*\}$,

(c) $\theta'(q) := \theta(q)$ für alle $q \in \text{Q}'$ und

(d) $\delta'(q, f) := \delta(q, f)$ für alle $q \in \mathrm{Q}', f \in \mathrm{Feat}$. $\Diamond$

Der Pfadwert liefert Unterstrukturen einer Feature Structure, weshalb man $F@\pi$ auch liest als „F unter π“. Ebenso wie die Funktion δ ist der Pfadwert für einen Pfad, der in der Feature Structure nicht existiert, undefiniert.

Ein Pfadwert ist in jedem Fall zusammenhängend, da nur Kanten übernommen werden, die vom neuen Wurzelknoten ausgehen.

Satz 2.20 [@-Operator erzeugt zusammenhängende Feature Structures] Sei F eine Feature Structure über einem Typsystem $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ und $\pi \in \mathrm{Feat}^*$ ein Pfad. Wenn $F@\pi$ definiert ist, dann ist $F@\pi$ eine zusammenhängende Feature Structure.

Beweis Sei $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle \mathrm{FS}_{\mathrm{TS}}$ und $F@\pi = \langle \mathrm{Q}', \bar{q}', \theta', \delta' \rangle \in \mathrm{FS}_{\mathrm{TS}}$ gemäß Definition 2.19. Sei $q' \in \mathrm{Q}'$, dann gilt es zu zeigen, daß es einen Pfad $\pi' \in \mathrm{Feat}^*$ gibt, so daß $\delta'(\bar{q}', \pi') = q'$. Nach Konstruktion von Q' in Definition 2.19 (b) gibt es zu jedem $q' \in \mathrm{Q}'$ einen Pfad $\pi' = f_1 f_2 \ldots f_n$ in F, so daß $q' = \delta(\bar{q}', \pi')$, also $\bar{q}' = q_0 \xrightarrow{f_1}_{\delta} q_1 \xrightarrow{f_2}_{\delta} q_2 \ldots \xrightarrow{f_n}_{\delta} q_n = q'$. Da nach 2.19 (b) auch für alle Präfixe $f_1 f_2 \ldots f_i$, $i \in \{1, \ldots, n\}$ von π' der entsprechende Knoten q_i in Q' aufgenommen wird, gilt $q_0, q_1, \ldots, q_n \in \mathrm{Q}'$. Nach 2.19 (d) gilt damit auch für alle $i \in \{1, \ldots, n\}$, daß $\delta'(q_{i-1}, f_i) = q_i$, also $q_{i-1} \xrightarrow{f_i}_{\delta'} q_i$, und damit $\delta'(\bar{q}', \pi') = \delta'(q_0, \pi') = \delta(q_0, \pi') = q_n = q'$. $\square$

Nicht zusammenhängende Feature Structures können durch den Pfadwert unter ihrer Wurzel zusammenhängend gemacht werden. Wir folgen Carpenter und betrachten im folgenden nur noch zusammenhängende Feature Structures.

Satz 2.21 [Neutrales Element des @-Operators]
Sei $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ ein Typsystem. Der leere Pfad ε stellt bezüglich des Pfadwerts von zusammenhängenden Feature Structures ein neutrales Element dar, es gilt für alle Feature Structures $F \in \mathrm{CFS}_{\mathrm{TS}}$:

$$F@\varepsilon = F$$

Beweis Sei $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle \in \mathrm{CFS}_{\mathrm{TS}}$ eine zusammenhängende Feature Structure und $F@\varepsilon = \langle \mathrm{Q}', \bar{q}', \theta', \delta' \rangle$. Es ist zu zeigen, daß folgt $F@\varepsilon = F$. Nach der Erweiterung von δ auf Pfade (Definition 2.16) gilt $\delta(\bar{q}, \varepsilon) = \bar{q}$, also ist $F@\varepsilon$ immer definiert und es gilt nach Definition 2.19 (a): $\bar{q}' = \bar{q}$. Die Definition von Q' in Definition 2.19 entspricht genau der von Q in Definition 2.18, wenn man $\bar{q}'$ durch $\bar{q}$, π durch ε und π' durch π substituiert, also ist $\mathrm{Q}' = \mathrm{Q}$. Die Gleichheit von θ' und θ und δ' und δ folgt direkt aus der Gleichheit von $\bar{q}'$ und $\bar{q}$ und Q' und Q. $\square$

Führen zwei Pfade in einer Feature Structure zu demselben Knoten, so sind auch die Pfadwerte gleich. Die Aussage gilt ebenso für die andere Richtung.

Satz 2.22 [Gleichheit von Knoten und Pfadwerten]
Sei TS $= \langle \text{Type}, \sqsubseteq, \text{Feat}, \text{approp} \rangle$ ein Typsystem, $F = \langle \text{Q}, \bar{q}, \theta, \delta \rangle \in \text{CFS}_{\text{TS}}$ eine zusammenhängende Feature Structure und seien $\pi, \pi' \in \text{Feat}^*$ Pfade. Dann gilt

$$\delta(\bar{q}, \pi) = \delta(\bar{q}, \pi') \Longleftrightarrow F@\pi = F@\pi'$$

Beweis Wir verwenden die Bezeichner aus dem Satz. Die Richtung von rechts nach links folgt direkt aus der Definition des Wurzelknotens des Pfadwerts, Definition 2.19 (a).
Für Richtung von links nach rechts ist zu zeigen, daß aus $\delta(\bar{q}, \pi) = \delta(\bar{q}, \pi')$ folgt $F@\pi = F@\pi'$. Sei $F_@ = \langle \text{Q}_@, \bar{q}_@, \theta_@, \delta_@ \rangle := F@\pi$, und $F'_@ = \langle \text{Q}'_@, \bar{q}'_@, \theta'_@, \delta'_@ \rangle := F@\pi'$. Wir zeigen die Behauptung durch $F_@ = F'_@$ und dies durch Gleichheit der Komponenten.
Nach Definition 2.19 ist $\bar{q}_@ = \delta(\bar{q}, \pi) = \delta(\bar{q}', \pi') = \bar{q}'_@$.
Für die Knotenmengen ergibt sich nach Definition 2.19

$$\begin{aligned} \text{Q}_@ &= \{\delta(\bar{q}_@, \tilde{\pi}) \mid \tilde{\pi} \in \text{Feat}^*\} \\ &= \{\delta(\bar{q}'_@, \tilde{\pi}) \mid \tilde{\pi} \in \text{Feat}^*\} \\ &= \text{Q}'_@. \end{aligned}$$

Die Gleichheit von $\theta_@$ und $\theta'_@$ sowie $\delta_@$ und $\delta'_@$ ergibt sich nach Definition 2.19 direkt aus der Gleichheit der Knotenmengen $\text{Q}_@$ und $\text{Q}'_@$. □

Der folgende Satz besagt, daß der Pfadwert unter konkatenierten Pfaden dem aufeinanderfolgenden Anwenden des Pfadwerts mit den Teilpfaden gleicht.

Satz 2.23 [Pfadwert und Pfad-Konkatenation]
Sei F eine Feature Structure über einem Typsystem TS $= \langle \text{Type}, \sqsubseteq, \text{Feat}, \text{approp} \rangle$ und seien $\pi, \pi' \in \text{Feat}^*$ Pfade. Wenn $F@(\pi\,\pi')$ definiert ist, dann gilt:

$$F@(\pi\,\pi') = (F@\pi)@\pi'$$

Beweis Sei $F = \langle \text{Q}, \bar{q}, \theta, \delta \rangle \in \text{FS}_{\text{TS}}$ und seien $F' = \langle \text{Q}', \bar{q}', \theta', \delta' \rangle := F@(\pi\,\pi')$, $F_@ = \langle \text{Q}_@, \bar{q}_@, \theta_@, \delta_@ \rangle := F@\pi$ und $F_{@@} = \langle \text{Q}_{@@}, \bar{q}_{@@}, \theta_{@@}, \delta_{@@} \rangle := F_@@\pi' = (F@\pi)@\pi'$. Wir zeigen durch die Gleichheit der Komponenten, daß $F' = F_{@@}$.
Durch die Konstruktion in Definition 2.19 können sich δ', $\delta_@$ und $\delta_{@@}$ und δ höchstens im Definitionsbereich unterscheiden, so daß alle durch δ substituiert werden können, wenn der jeweilige Knoten Element der entsprechenden Knotenmenge ist.
Nach Definition 2.19 (b) gilt $\text{Q}' = \{\delta(\bar{\text{q}}, \pi\,\pi'\,\pi'') \mid \pi'' \in \text{Feat}^*\}$ sowie $\text{Q}_@ = \{\delta(\bar{\text{q}}, \pi\,\pi'') \mid \pi'' \in \text{Feat}^*\}$ und damit für $\text{Q}_{@@}$

$$\begin{aligned} \text{Q}_{@@} &= \{\delta_@(\bar{q}_@, \pi'\,\pi'') \mid \pi'' \in \text{Feat}^*\} \\ &= \{\delta_@(\delta(\bar{q}, \pi), \pi'\,\pi'') \mid \pi'' \in \text{Feat}^*\} \end{aligned}$$

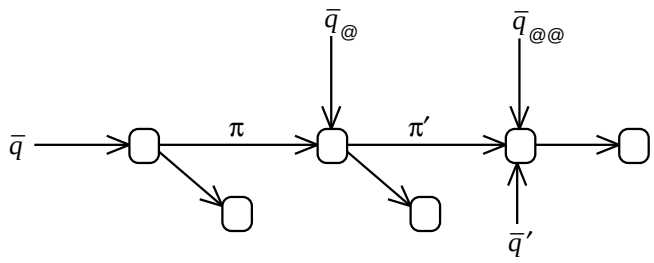

Abbildung 2.16: Skizze zum Beweis von Satz 2.23.

$$\begin{aligned} &= \{\delta(\delta(\bar{q},\pi),\pi'\,\pi'') \mid \pi'' \in \mathrm{Feat}^*\} \\ &= \{\delta(\bar{q},\pi\,\pi'\,\pi'') \mid \pi'' \in \mathrm{Feat}^*\} \\ &= \mathrm{Q}'. \end{aligned}$$

Für die Wurzelknoten ergibt sich $\bar{q}_{@@} = \delta_{@}(\bar{q}_{@},\pi') = \delta_{@}(\delta(\bar{q},\pi),\pi') = \delta(\delta(\bar{q},\pi),\pi') = \delta(\bar{q},\pi\,\pi') = \bar{q}'$.
Wie oben bereits begründet folgt aus $\mathrm{Q}_{@@} = \mathrm{Q}'$ die Gleichheit von $\theta_{@@}$ und θ' sowie $\delta_{@@}$ und δ', da sich diese nach Konstruktion nur im Definitionsbereich unterscheiden könnten. □

Das Gegenstück zum Pfadwert stellt der Vorpfad-Operator :: dar, der eine Feature Structure aus einem Pfad und einer gegebenen Feature Structure konstruiert, indem der Pfad vor die Feature Structure geschrieben wird. Der Vorpfad-Operator erweist sich in der Konstruktion von komplexen Feature Structures als nützlich, wie sie für die Definition von FSNets in Kapitel 4 benötigt werden. Der Operator stammt nicht von Carpenter, sondern wird hier neu eingeführt. Aus diesem Grunde werden im folgenden vor allem weitere Sätze aufgestellt und bewiesen, die Eigenschaften des Vorpfads beschreiben.

Ein Vorpfad wird in zwei Schritten eingeführt: Wie bei der Pfad-Knoten-Funktionen wird zunächst der Vorpfad für ein Feature definiert (also eigentlich ein „Vorfeature“), dann auf Pfade erweitert. Die Definition wird so gewählt, daß ein eindeutiges Ergebnis entsteht. Um einen eindeutigen neuen Knoten zu konstruieren wird hier von einer üblichen Mengentheorie ausgegangen, in der Mengen nicht sich selbst als Element enthalten können.

Definition 2.24 [Vorpfad / ::-Operator]
Sei $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp}\rangle$ ein Typsystem. Der *Vorpfad* einer Feature Structure zu einem Pfad sei definiert durch den *Vorpfad-Operator* $(::) : \mathrm{Feat}^* \times \mathrm{FS}_{\mathrm{TS}} \rightarrow \mathrm{FS}_{\mathrm{TS}}$.

Für $f \in \text{Feat}$, also Pfade der Länge 1, sei

$$\begin{aligned} f::F := (::)(f,F) &:= \langle \mathrm{Q} \cup \{\mathrm{Q}\}, \mathrm{Q}, \theta \cup \theta_{\mathrm{Q}}, \delta \cup \delta_{\mathrm{Q}} \rangle \\ \text{mit } \theta_{\mathrm{Q}}(\mathrm{Q}) &:= \top \\ \text{und } \delta_{\mathrm{Q}}(\mathrm{Q}, f) &:= \bar{q}, \\ \theta_{\mathrm{Q}}\text{und } \delta_{\mathrm{Q}} & \quad \text{undefiniert sonst.} \end{aligned}$$

Für alle anderen Pfade sei

$$\begin{aligned} \varepsilon::F &:= F \\ \text{und } \forall \pi' \in \text{Feat}^+, f' \in \text{Feat} : (f'\,\pi')::F &:= f'::(\pi'::F). \end{aligned}$$

$\Diamond$

Den Ausdruck $\pi::F$ liest man als „π vor F“. Auch für den Vorpfad-Operator ergibt sich nach Definition der leere Pfad als neutrales Element.

Satz 2.25 [Neutrales Element des ::-Operators]
Der leere Pfad ε stellt bezüglich des Vorpfad-Operators für beliebige Feature Structures ein neutrales Element dar, es gilt für alle Feature Structures F:

$$\varepsilon::F = F. \qquad \square$$

Es gibt eine zu Satz 2.23 analoge Eigenschaft des Vorpfads bezüglich konkatenierten Pfaden.

Satz 2.26 [Vorpfad und Pfad-Konkatenation]
Sei F eine zusammenhängende Feature Structure über einem Typsystem $\text{TS} = \langle \text{Type}, \sqsubseteq, \text{Feat}, \text{approp} \rangle$ und seien $\pi, \pi' \in \text{Feat}^*$ Pfade, dann gilt:

$$(\pi\,\pi')::F = \pi::(\pi'::F)$$

Beweis Sei $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle$ eine zusammenhängende Feature Structure über $\text{TS} = \langle \text{Type}, \sqsubseteq, \text{Feat}, \text{approp} \rangle$ sowie $\pi = f_1\, f_2 \ldots f_n \in \text{Feat}^*$ und $\pi' \in \text{Feat}^*$ Pfade.
Wir zeigen die Behauptung durch Induktion über die Länge n des Pfads π. Dafür zeigen wir zunächst, daß gilt:

$$\forall f \in \text{Feat}, \pi'' \in \text{Feat}^* : (f\,\pi'')::F = f::(\pi''::F) \tag{2.6}$$

Nach Definition 2.24 gilt Behauptung 2.6 für $\pi'' = \varepsilon$, da nach Satz 2.25 der Pfad ε das neutrale Element des Vorpfads darstellt und somit $(f\,\varepsilon)::F = f::F = f::(\varepsilon::F)$.
Für die übrigen Pfade $\pi'' = \hat{f}\,\hat{\pi}$ gilt die Behauptung trivialerweise aus der in Definition 2.24 gegebenen Erweiterung auf Pfade.
Sei $\pi_0 = \varepsilon$ und für $i \in \{0, \ldots, n-1\}$ sei $\pi_{i+1} := f_{n-i}\,\pi_i$. Wir zeigen $(\pi_k\,\pi')::F = \pi_k::(\pi'::F)$ für alle $k \in \{0, \ldots, n\}$.

Für $k = 0$ ist

$$\begin{array}{rcll}(\pi_0\,\pi')::F & = & (\varepsilon\,\pi')::F & \\ & = & \pi'::F & \\ & = & \varepsilon::(\pi'::F) & \text{(nach Satz 2.25)} \\ & = & \pi_0::(\pi'::F) & \end{array}$$

Sei $(\pi_i\,\pi')::F = \pi_i::(\pi'::F)$ für alle $i \in \{0, \ldots, k\}$ gezeigt. Dann ist

$$\begin{array}{rcll}(\pi_{k+1}\,\pi')::F & = & (f_{n-k}\,\pi_k\,\pi')::F & \\ & = & f_{n-k}::(\pi_k\,\pi'::F) & \text{(nach Aussage 2.6)} \\ & = & f_{n-k}::(\pi_k::(\pi'::F)) & \\ & = & (f_{n-k}\,\pi_k)::(\pi'::F) & \text{(nach Aussage 2.6)} \\ & = & \pi_{k+1}::(\pi'::F) & \end{array}$$

Für $k = n$ ist damit die Behauptung gezeigt. □

Es wurde oben gesagt, daß ::- und @-Operator Gegenstücke zueinander darstellen. Diese Beobachtung läßt sich in zwei Sätzen formal festhalten.

Satz 2.27 [Zusammenhang zwischen Pfadwert und Vorpfad (1)]
Sei $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp}\rangle$ ein Typsystem. Es gilt für alle $\pi \in \mathrm{Feat}^*$ und alle zusammenhängenden Feature Structures $F \in CFS_{TS}$:

$$(\pi::F)@\pi = F$$

Beweis Sei $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta\rangle$ eine zusammenhängende Feature Structure über TS und $\pi = f_1\,f_2 \ldots f_n \in \mathrm{Feat}^*$ ein Pfad. Wir zeigen die Behauptung durch Induktion über n. Dafür zeigen wir zunächst, daß gilt

$$\forall f \in \mathrm{Feat} : (f::F)@f = F \tag{2.7}$$

Sei $F_f = \langle \mathrm{Q}_f, \bar{q}_f, \theta_f, \delta_f\rangle := f::F$ und $F_@ = \langle \mathrm{Q}_@, \bar{q}_@, \theta_@, \delta_@\rangle$. $:= (f::F)@f = F_f@f$. Es ist dann zu zeigen, daß $F_@ = F$. Wir zeigen die Gleichheit der einzelnen Komponenten.
Nach Definition 2.24 ist $\mathrm{Q}_f = \mathrm{Q} \cup \{\mathrm{Q}\}$, $\bar{q}_f = \mathrm{Q}$, $\theta_f = \theta \cup \theta_\mathrm{Q}$ mit $\theta_\mathrm{Q}(\mathrm{Q}) = \top$ und $\delta_f = \delta \cup \delta_\mathrm{Q}$ mit $\delta_\mathrm{Q}(\mathrm{Q}, f) = \bar{q}$, δ_Q undefiniert sonst.
Nach Definition 2.19 (a) ist $\bar{q}_@ = \delta_f(\mathrm{Q}, f) = (\delta \cup \delta_\mathrm{Q})(\mathrm{Q}, f) = \delta_\mathrm{Q}(\mathrm{Q}, f) = \bar{q}$.
Da F zusammenhängend ist, gilt nach Definition 2.18

$$\mathrm{Q} = \{\delta(\bar{q}, \pi) \mid \pi \in \mathrm{Feat}^*\} \tag{2.8}$$

Da $\delta_f(\mathrm{Q}, f) = \bar{q}$ die einzige neue Kante in F_f gegenüber F ist und F zusammenhängend ist, sind auch die Erweiterungen von δ_f und δ auf Pfade für den eingeschränkten Definitionsbereich $\mathrm{Q} \times \mathrm{Feat}^*$ gleich:

$$\forall q \in \mathrm{Q}, \pi' \in \mathrm{Feat}^* : \delta_f(\bar{q}, \pi') = q \Leftrightarrow \delta(\bar{q}, \pi') = q \tag{2.9}$$

Nach Definition 2.19 (a) ist

$$\begin{array}{rcll} \mathrm{Q}_@ & = & \{\delta_f(\bar{q}_@, \pi') \mid \pi' \in \mathrm{Feat}^*\} & \\ & = & \{\delta_f(\bar{q}, \pi') \mid \pi' \in \mathrm{Feat}^*\} & \\ & = & \{\delta(\bar{q}, \pi') \mid \pi' \in \mathrm{Feat}^*\} & \text{(nach Aussage 2.9, da } \bar{q} \in \mathrm{Q}) \\ & = & \mathrm{Q} & \text{(nach Aussage 2.8)} \end{array}$$

Aus $\bar{q}_@ = \bar{q}$ und $\mathrm{Q}_@ = \mathrm{Q}$ folgt die Gleichheit von $\delta_@$ und δ sowie $\theta_@$ und θ direkt aus den Definitionen 2.24 und 2.19, in denen die Funktionswerte für den ursprünglichen Definitionsbereich übernommen werden.
Damit ist Behauptung 2.7 gezeigt. Es bleibt die Erweiterung auf Pfade zu zeigen. Wir zeigen die Behauptung $(\pi::F)@\pi = F$ durch Induktion über die Länge n des Pfads π.
Sei $\pi_0 = \varepsilon$ und für $i \in \{0, \ldots, n-1\}$ sei $\pi_{i+1} := f_{n-i}\,\pi_i$. Wir zeigen $(\pi_k::F)@\pi_k = F$ für alle $k \in \{0, \ldots, n\}$.
Für $k = 0$ ist

$$\begin{array}{rcll} (\pi_k::F)@\pi_k & = & (\pi_0::F)@\pi_0 & \\ & = & (\varepsilon::F)@\varepsilon & \\ & = & F@\varepsilon & \text{(nach Satz 2.25)} \\ & = & F & \text{(nach Satz 2.21)} \end{array}$$

Sei $(\pi_i::F)@\pi_i = F$ für alle $i \in \{0, \ldots, k\}$ gezeigt. Dann ist

$$\begin{array}{rcll} (\pi_{k+1}::F)@\pi_{k+1} & = & (f_{n-k}\,\pi_k::F)@(f_{n-k}\,\pi_k) & \\ & = & (f_{n-k}::(\pi_k::F))@(f_{n-k}\,\pi_k) & \text{(nach Satz 2.26)} \\ & = & ((f_{n-k}::(\pi_k::F))@f_{n-k})@\pi_k & \text{(nach Satz 2.23)} \\ & = & (\pi_k::F)@\pi_k & \text{(nach Aussage 2.7)} \\ & = & F & \end{array}$$

Für $k = n$ ist damit die Behauptung gezeigt. □

Satz 2.28 [Zusammenhang zwischen Pfadwert und Vorpfad (2)]
Sei $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ ein Typsystem. Es gilt für alle $\pi \in \mathrm{Feat}^*$ und alle Feature Structures $F \in FS_{TS}$, für die $F@\pi$ definiert ist:

$$\pi::(F@\pi) \sqsubseteq F$$

Beweis Sei $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle$ eine zusammenhängende Feature Structure über TS und $\pi = f_1\, f_2 \ldots f_n \in \mathrm{Feat}^*$ ein Pfad. Die Behauptung kann wie im vorherigen Beweis durch Induktion über n gezeigt werden, wenn wir beweisen können, daß gilt

$$\forall f \in \mathrm{Feat} : F@f \text{ ist definiert } \Rightarrow f::(F@f) \sqsubseteq F \tag{2.10}$$

Sei $F_@ = \langle \mathrm{Q}_@, \bar{q}_@, \theta_@, \delta_@ \rangle := F@f$ und $F_f = \langle \mathrm{Q}_f, \bar{q}_f, \theta_f, \delta_f \rangle := f::F_@ = f::(F@f)$. Es ist dann zu zeigen, daß $F_f \sqsubseteq F$. Dafür zeigen wir die Existenz einer Funktion $\mathrm{h} : \mathrm{Q}_f \to Q$, für welche die Eigenschaften (a) bis (c) aus Definition 2.29 gelten.

Nach Definition 2.19 ist $Q_@ \subseteq Q$ und nach Definition 2.24 ist $Q_f = Q_@ \cup \{\bar{q}_f\}$ und $\bar{q}_f = Q_@$. Sei $h(\bar{q}_f) := \bar{q}$ und $\forall q \in Q_@ : h(q) := q$.
Eigenschaft (a) aus Definition 2.29 ergibt sich direkt aus der Definition von h.
Für Eigenschaft (b) werden zwei Fälle unterschieden. Für $q \in Q_@$ gilt $\theta(h(q)) = \theta(q) = \theta_f(q) \Rightarrow \theta_f(q) \sqsubseteq \theta(h(q))$. Für $q = \bar{q}_f$ gilt $\theta_f(\bar{q}_f) = \top \sqsubseteq \theta(h(\bar{q}_f)) = \theta(\bar{q})$.
Für Eigenschaft (c) ist zu zeigen, daß für alle $f' \in Feat, q_1, q_2 \in Q_f : \delta_f(q_1, f') = q_2 \Rightarrow \delta(h(q_1), f') = h(q_2)$. Wiederum betrachten wir zunächst den Fall $q_1, q_2 \in Q_@$. Hier gilt die Behauptung wegen $h(q_1) = q_1$ sowie $h(q_2) = q_2$ und $\delta_f(q_1, f') = \delta(q_1, f')$. Der einzige Knoten, der nicht in $Q_@$ liegt, ist $\bar{q}_f$, und von diesem geht nach Konstruktion in Definition 2.24 nur die Kante $\delta_f(\bar{q}_f, f) = \bar{q}_@$ aus. Mit $h(\bar{q}_f) = \bar{q}$ und $h(\bar{q}_@) = \bar{q}_@$ muß demnach die Existenz einer Kante $\delta(\bar{q}, f) = \bar{q}_@$ gezeigt werden, die nach der Konstruktion in Definition 2.19 gegeben ist, da $\bar{q}_@ = \delta(\bar{q}, f)$. □

Nach diesen grundlegenden Definitionen für Feature Structures wenden wir uns nun dem Problem zu, wie Subsumption und Unifikation von Typen auf Feature Structures übertragen werden können.

2.3.3 Subsumption und Unifikation von Feature Structures

In diesem Abschnitt soll gezeigt werden, wie Subsumption und Join-Bildung, hier als Unifikation bezeichnet, von Typen auf Feature Structures übertragen werden.

Eine Subsumptionsrelation über Feature Structures

Subsumption findet beispielsweise Anwendung beim Formulieren von Anfragen an eine Menge von Datenobjekten. Jene Datenobjekte werden ausgewählt, die von der Anfrage subsumiert werden. Damit verfügt man über eine einheitliche Darstellung für "konkrete" Information, wie sie sich z.B. in einem Katalog befinden könnte, und "abstrakte" Information, die unterspezifiziert ist, um eine Menge von Objekten zu beschreiben. Unifikation hingegen dient zur Konsistenzprüfung von Informationsobjekten und zur exakten Vereinigung konsistenter Informationsobjekte. Dies wird anwendungsorientiert motiviert durch das Problem der Zusammenführung verteilter Kopien von Informationsobjekten (siehe Abschnitt 1.1.1).

Eine wichtige Voraussetzung für das intuitive Verständnis der folgenden Definitionen ist die oben diskutierte Sichtweise von Feature Structures als unterspezifizierte Beschreibungen von Objekten.

In diesem Sinne können wir bei Feature Structures wie bei Typen von Subsumption sprechen: Eine Feature Structure F subsumiert eine Feature Structure F', wenn F' zusätzliche (genauer: mindestens die gleiche) Information enthält. Es gibt drei frei kombinierbare Möglichkeiten, den Informationsgehalt einer Feature Structure zu erhöhen:

(a) Es wird ein Feature (eine Kante im Graphen) und dabei unter Umständen ein Wert (neuer Knoten) hinzugefügt.

(b) Der Typ eines bestehenden Werts (Knotens) wird genauer spezifiziert.

(c) Werte werden identifiziert (Knoten werden vereinigt / fallen zusammen).

Der letzte Punkt benötigt etwas Erläuterung: Man könnte annehmen, daß das Identifizieren von Knoten im Gegenteil einen Informations*verlust* bedeutet. Dies ist aber nicht der Fall, da unterschiedliche Knoten in einer Feature Structure bedeuten, daß über deren Gleichheit nichts bekannt ist. Durch die Identifizierung ist die Identität beider Knoten bekannt, so daß dies tatsächlich einen Informationsgewinn bedeutet. Selbst, wenn beide Knoten denselben Typ und dieselben Features mit denselben Unterstrukturen besitzen, ist die Information, daß es sich um ein und denselben Knoten handelt, spezifischer. Carpenter stellt in [Carpenter 1992] eine Art der Negation vor, mit der über zwei Knoten ausgesagt werden kann, daß diese nie identifiziert werden können, was auch einen Informationsgewinn bedeutet (siehe dazu die Diskussion von Negation und extensionalen Typen in [Carpenter 1992]).

Folgende Definition von Carpenter formalisiert, wie der Informationsgehalt von Feature Structures verglichen werden kann. Dazu wird wie bei Typen der Begriff der Subsumption verwendet: Eine Feature Structure F subsumiert F', wenn F' mindestens die Information von F enthält.

Definition 2.29 [Feature-Structure-Subsumption]
Seien $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ ein Typsystem und $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle$ und $F' = \langle \mathrm{Q}', \bar{q}', \theta', \delta' \rangle$ Feature Structures über TS. F subsumiert F', $F \sqsubseteq F'$, genau dann, wenn es eine totale Funktion $\mathrm{h} : \mathrm{Q} \to \mathrm{Q}'$ gibt, so daß:

(a) $\mathrm{h}(\bar{q}) = \bar{q}'$ (Erhalt des Wurzelknotens),

(b) für alle $q \in \mathrm{Q} : \theta(q) \sqsubseteq \theta'(\mathrm{h}(q))$ (mind. so spezielle Typen) und

(c) für alle $f \in \mathrm{Feat}, q_1, q_2 \in \mathrm{Q} : \delta(q_1, f) = q_2 \Rightarrow \delta'(\mathrm{h}(q_1), f) = \mathrm{h}(q_2)$
(Kanten bleiben erhalten). ♢

Man beachte, daß durch die Definitionen 2.29 und 2.1 das Relationssymbol $\sqsubseteq$ überladen ist, da es sowohl für Typ- als auch für Feature-Structure-Subsumption eingesetzt wird. Anhand der jeweiligen Argumente ist die Zuordnung eindeutig zu erkennen.

Der folgende Satz besagt, daß sich die Subsumption auf Pfadwerte überträgt.

Satz 2.30 [Übertragung der Subsumption auf Pfadwerte]
Seien F, F' Feature Structures über dem Typsystem $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ mit $F \sqsubseteq F'$ und $\pi \in \mathrm{Feat}^*$ ein Pfad, für den $F@\pi$ definiert ist. Dann gilt $F@\pi \sqsubseteq F'@\pi$.

Beweis Seien $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle, F' = \langle \mathrm{Q}', \bar{q}', \theta', \delta' \rangle$, $F, F' \in \mathrm{FS}_{\mathrm{TS}}$, $F \sqsubseteq F'$ und sei $\pi \in \mathrm{Feat}^*$ ein Pfad, für den $F@\pi = \langle \mathrm{Q}_@, \bar{q}_@, \theta_@, \delta_@ \rangle$ definiert ist, sowie

$F'@\pi = \langle \mathrm{Q}'_@, \bar{q}'_@, \theta'_@, \delta'_@ \rangle$. $F'@\pi$ ist definiert, da aus ($F@\pi$ ist definiert) nach Definition 2.19 folgt, daß ($\delta(\bar{q}, \pi)$ ist definiert) und sich für $F \sqsubseteq F'$ leicht zeigen läßt, daß $\delta(\bar{q}, \pi)$ ist definiert $\Rightarrow \delta'(\bar{q}', \pi)$ ist definiert ([Carpenter 1992]).
Aus $F \sqsubseteq F'$ folgt nach Definition 2.29 die Existenz einer Funktion $\mathrm{h} : \mathrm{Q} \to \mathrm{Q}'$, welche die dort genannten Eigenschaften (a) bis (c) besitzt. Zu zeigen ist die Existenz einer Subsumptions-Funktion $\mathrm{h}_@ : \mathrm{Q}_@ \to \mathrm{Q}'_@$, die ebenso die Eigenschaften aus Definition 2.29 (a) bis (c) besitzt.
Zunächst wird gezeigt, daß für Urbilder aus $\mathrm{Q}_@ \subseteq \mathrm{Q}$ die Funktionswerte von h in $\mathrm{Q}'_@ \subseteq \mathrm{Q}'$ liegen müssen, also daß

$$\forall q \in \mathrm{Q}_@ : \mathrm{h}(q) \in \mathrm{Q}'_@ \tag{2.11}$$

Da $F@\pi$ nach Satz 2.20 zusammenhängend ist, gibt es für jedes $q \in \mathrm{Q}_@$ einen Pfad $\hat{\pi} = f_1, f_2, \ldots, f_n$, für den gilt: $\bar{q}_@ \xrightarrow[\delta_@]{f_1} q_1 \xrightarrow[\delta_@]{f_2} q_2 \ldots \xrightarrow[\delta_@]{f_n} q_n = q,\ q_1, q_2, \ldots q_n \in \mathrm{Q}_@$. Nach Definition 2.29 (c) werden durch h alle Kanten auf F' übertragen. Also gibt es den Pfad $\hat{\pi}$ auch in F' mit $\mathrm{h}(\bar{q}_@) \xrightarrow[\delta'_@]{f_1} \mathrm{h}(q_1) \xrightarrow[\delta'_@]{f_2} \mathrm{h}(q_2) \ldots \xrightarrow[\delta'_@]{f_n} \mathrm{h}(q)$. Da $\mathrm{h}(\bar{q}_@) = \mathrm{h}(\delta(\bar{q}, \pi)) = \delta'(\bar{q}', \pi) = \bar{q}'_@$, also $\mathrm{h}(\bar{q}_@) = \bar{q}'_@ \in \mathrm{Q}'_@$ und auch $F'@\pi$ nach Satz 2.20 zusammenhängend ist, folgt $\mathrm{h}(q_1), \mathrm{h}(q_2), \ldots, \mathrm{h}(q_n) \in \mathrm{Q}'_@$, also $\mathrm{h}(q) \in \mathrm{Q}'_@$.
Da nach Definition 2.19 gilt $\mathrm{Q}_@ \subseteq \mathrm{Q}$, kann $\mathrm{h}_@$ als Einschränkung von h auf $\mathrm{Q}_@$ definiert werden, also $\forall q \in \mathrm{Q}_@ : \mathrm{h}_@(q) := \mathrm{h}(q)$. Durch den Beweis der Behauptung 2.11 ist sichergestellt, daß die Funktionswerte von $\mathrm{h}_@$ in $\mathrm{Q}_@$ liegen.
Es sollen nun die Eigenschaften (a) bis (c) aus Definition 2.29 für $\mathrm{h}_@$ gezeigt werden. Eigenschaft (a) folgt aus $\mathrm{h}_@(\bar{q}_@) = \mathrm{h}(\bar{q}_@) = \bar{q}'_@$.
Eigenschaft (b) ergibt sich folgendermaßen:

$$\begin{aligned} & \forall q \in \mathrm{Q} : \theta(q) \sqsubseteq \theta'(\mathrm{h}(q)) \\ \Rightarrow\ & \forall q \in \mathrm{Q}_@ : \theta(q) \sqsubseteq \theta'(\mathrm{h}_@(q)) \\ \Rightarrow\ & \forall q \in \mathrm{Q}_@ : \theta_@(q) \sqsubseteq \theta'_@(\mathrm{h}_@(q)) \end{aligned}$$

Für Eigenschaft (c) ergibt sich:

$$\begin{aligned} & \forall f \in \mathrm{Feat}, q_1, q_2 \in \mathrm{Q} : \delta(q_1, f) = q_2 \Rightarrow \delta'(\mathrm{h}(q_1), f) = \mathrm{h}(q_2) \\ \Rightarrow\ & \forall f \in \mathrm{Feat}, q_1, q_2 \in \mathrm{Q}_@ : \delta(q_1, f) = q_2 \Rightarrow \delta'(\mathrm{h}_@(q_1), f) = \mathrm{h}_@(q_2) \\ \Rightarrow\ & \forall f \in \mathrm{Feat}, q_1, q_2 \in \mathrm{Q}_@ : \delta_@(q_1, f) = q_2 \Rightarrow \delta'_@(\mathrm{h}_@(q_1), f) = \mathrm{h}_@(q_2) \end{aligned}$$

Damit ist gezeigt, daß gilt $F@\pi \sqsubseteq F'@\pi$. □

Auch zu diesem Satz gibt es ein Gegenstück mit dem Vorpfad-Operator. Hier überträgt sich die Subsumption sogar für beliebige Pfade in beide Richtungen.

Satz 2.31 [Übertragung der Subsumption auf Vorpfade]
Seien F, F' zusammenhängende Feature Structures über dem Typsystem $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ und $\pi \in \mathrm{Feat}^*$ ein Pfad. Dann gilt

$$F \sqsubseteq F' \iff \pi::F \sqsubseteq \pi::F'$$

Beweis Sei $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle$ und $F' = \langle \mathrm{Q}', \bar{q}', \theta', \delta' \rangle \in \mathrm{FS}_{\mathrm{TS}}$.
Wir zeigen die Behauptung für $f \in \mathrm{Feat}$:

$$F \sqsubseteq F' \Longleftrightarrow f{::}F \sqsubseteq f{::}F'$$

Die im Satz behauptete Erweiterung auf Pfade ergibt sich analog zum Beweis von Satz 2.26 durch Induktion über die Länge des Pfads und wird hier nicht gezeigt.
Sei $F_f = \langle \mathrm{Q}_f, \bar{q}_f, \theta_f, \delta_f \rangle := f{::}F$ und $F'_f = \langle \mathrm{Q}'_f, \bar{q}'_f, \theta'_f, \delta'_f \rangle := f{::}F'$. Aus Definition 2.24 folgt, daß $\bar{q}_f = \mathrm{Q}$, $\mathrm{Q}_f = \mathrm{Q} \cup \{\mathrm{Q}\} = \mathrm{Q} \cup \{\bar{q}_f\}$, $\theta_f = \theta \cup \theta_{\mathrm{Q}}$ mit $\theta_{\mathrm{Q}}(\bar{q}_f) = \top$ und $\delta_f = \delta \cup \delta_{\mathrm{Q}}$ mit $\delta_{\mathrm{Q}}(\bar{q}_f, f) = \bar{q}$. Analog folgt für F'_f, daß $\bar{q}'_f = \mathrm{Q}'$, $\mathrm{Q}'_f = \mathrm{Q}' \cup \{\mathrm{Q}'\} = \mathrm{Q}' \cup \{\bar{q}'_f\}$, $\theta'_f = \theta' \cup \theta'_{\mathrm{Q}}$ mit $\theta'_{\mathrm{Q}}(\bar{q}'_f) = \top$ und $\delta'_f = \delta' \cup \delta'_{\mathrm{Q}}$ mit $\delta'_{\mathrm{Q}}(\bar{q}'_f, f) = \bar{q}'$.
Wir zeigen zunächst die Richtung von links nach rechts, also

$$F \sqsubseteq F' \Rightarrow F_f \sqsubseteq F'_f$$

Aus $F \sqsubseteq F'$ folgt nach Definition 2.29 die Existenz einer Funktion $\mathrm{h} : \mathrm{Q} \to \mathrm{Q}'$, welche die dort genannten Eigenschaften (a) bis (c) besitzt. Zu zeigen ist die Existenz einer Subsumptions-Funktion $\mathrm{h}_f : \mathrm{Q}_f \to \mathrm{Q}'_f$, die ebenso die Eigenschaften 2.29 (a) bis (c) besitzt.
Für alle $q \in \mathrm{Q}$ sei $\mathrm{h}_f(q) := \mathrm{h}(q)$ und für $\bar{q}_f = \mathrm{Q}$ sei $\mathrm{h}_f(\bar{q}_f) := \bar{q}'_f = \mathrm{Q}'$. Für diese Funktion h_f sind nun die Eigenschaften (a) bis (c) aus Definition 2.29 zu zeigen.
Eigenschaft (a) folgt direkt aus der Definition von $\mathrm{h}_f(\bar{q}_f) = \bar{q}'_f$.
Für (b) ist zu zeigen, daß $\forall q \in \mathrm{Q}_f : \theta_f(q) \sqsubseteq \theta'_f(\mathrm{h}_f(q))$. Wir treffen die Fallunterscheidungen $q \in \mathrm{Q}$ und $q = \mathrm{Q} = \bar{q}_f$. Für $q \in \mathrm{Q}$ ist $\mathrm{h}_f(q) = \mathrm{h}(q)$ und nach Definition 2.24 gilt $\theta_f(q) = \theta(q)$ sowie $\theta'_f(q) = \theta'(q)$. Die Behauptung folgt durch Einsetzen in die durch Eigenschaft 2.29 (b) für h geltende Formel $\theta(q) \sqsubseteq \theta'(\mathrm{h}(q))$. Für den anderen Fall $q = \mathrm{Q} = \bar{q}_f$ ist nach Definition 2.24 $\theta_f(\mathrm{Q}) = \top = \theta'_f(\mathrm{Q}') = \theta'_f(\mathrm{h}_f(\mathrm{Q}))$, woraus die Behauptung folgt.
Für (c) ist zu zeigen, daß für alle $f' \in \mathrm{Feat}, q_1, q_2 \in \mathrm{Q}_f : \delta_f(q_1, f') = q_2 \Rightarrow \delta'_f(\mathrm{h}_f(q_1), f') = \mathrm{h}_f(q_2)$. Wir treffen wie oben die Fallunterscheidungen $q_1 = \mathrm{Q} = \bar{q}_f$ und $q_1 \in \mathrm{Q}$.
Für $q_1 = \mathrm{Q} = \bar{q}_f$ gibt es nur eine Kante in F_f, also folgt $f' = f$ und $q_2 = \bar{q}$. Aus $\delta'_f(\bar{q}'_f, f) = \bar{q}'$ folgt mit $\bar{q}'_f = \mathrm{h}_f(\bar{q}_f)$ und $\bar{q}' = \mathrm{h}(\bar{q}) = \mathrm{h}_f(\bar{q})$, daß $\delta'_f(\mathrm{h}_f(\bar{q}_f), f) = \mathrm{h}_f(\bar{q})$. Durch Einsetzen von $f = f'$ und $\bar{q} = q_2$ erhält man die zu zeigende Behauptung.
Weiterhin gilt nach Definition von δ_f:

$$\delta_f(q_1, f') = q_2 \Longleftrightarrow \delta(q_1, f') = q_2 \tag{2.12}$$

Aus der Definition von δ folgt, daß $q_2 \in \mathrm{Q}$.
Für alle $q'_1 \in \mathrm{Q}', q'_2 \in \mathrm{Q}'_f$ gilt analog zu oben:

$$\delta'_f(q'_1, f') = q'_2 \Longleftrightarrow \delta'(q'_1, f') = q'_2 \tag{2.13}$$

Es gilt demnach:

$$\begin{array}{lll} & \delta_f(q_1, f') = q_2 & \\ \Rightarrow & \delta(q_1, f') = q_2 & \text{(nach Aussage 2.12)} \\ \Rightarrow & \delta'(\mathrm{h}(q_1), f') = \mathrm{h}(q_2) & \text{(nach Definition 2.29 (c) für h)} \\ \Rightarrow & \delta'_f(\mathrm{h}(q_1), f') = \mathrm{h}(q_2) & \text{(nach Aussage 2.13)} \\ \Rightarrow & \delta'_f(\mathrm{h}_f(q_1), f') = \mathrm{h}_f(q_2) & \text{(nach Definition von } \mathrm{h}_f(q) \text{ für } q \in \mathrm{Q}) \end{array}$$

Damit ist die Behauptung gezeigt, daß $F \sqsubseteq F' \Rightarrow F_f \sqsubseteq F'_f$.
Wir zeigen nun die Richtung von rechts nach links, also

$$F \sqsubseteq F' \Leftarrow F_f \sqsubseteq F'_f$$

Es werden dieselben Bezeichner wie im ersten Teil des Beweises verwendet.
Aus $F_f \sqsubseteq F'_f$ folgt nach Definition 2.29 die Existenz einer Funktion $\mathrm{h}'_f : \mathrm{Q}_f \to \mathrm{Q}'_f$, welche die dort genannten Eigenschaften (a) bis (c) besitzt. Zu zeigen ist die Existenz einer Subsumptions-Funktion $\mathrm{h}' : \mathrm{Q} \to \mathrm{Q}'$, die ebenso die Eigenschaften 2.29 (a) bis (c) besitzt.
Zunächst wird gezeigt, daß für Urbilder aus $\mathrm{Q} \subset \mathrm{Q}_f$ die Funktionswerte von h'_f in $\mathrm{Q}' \subset \mathrm{Q}'_f$ liegen müssen, also daß

$$\forall q \in \mathrm{Q} : \mathrm{h}'_f(q) \in \mathrm{Q}'$$

Da $\mathrm{h}'_f(q) \in \mathrm{Q}'_f$ und $\mathrm{Q}'_f = \mathrm{Q}' \cup \{\bar{q}'_f\}$, ist zu zeigen, daß

$$\begin{array}{rll} \forall q \in \mathrm{Q} : & \mathrm{h}'_f(q) & \notin \{\bar{q}'_f\} \\ \Leftrightarrow \forall q \in \mathrm{Q} : & \mathrm{h}'_f(q) & \neq \bar{q}'_f \end{array} \tag{2.14}$$

Wir nehmen an, daß es ein $\hat{q} \in \mathrm{Q}$ gibt, für das gilt $\mathrm{h}'_f(\hat{q}) = \bar{q}'_f$ und zeigen den Widerspruch. Dafür treffen wir die Fallunterscheidung $\hat{q} = \bar{q}$ und $\hat{q} \in \mathrm{Q} - \{\bar{q}\}$.
Für $\hat{q} = \bar{q}$ ist $\mathrm{h}'_f(\hat{q}) = \mathrm{h}'_f(\bar{q}) = \bar{q}'$ und nach Annahme $\mathrm{h}'_f(\hat{q}) = \bar{q}'_f = \mathrm{Q}'$, also $\bar{q}' = \mathrm{Q}'$. Wegen $\bar{q}' \in \mathrm{Q}'$ folgt $\mathrm{Q}' \in \mathrm{Q}'$, was in der hier verwendeten üblichen Mengentheorie nicht möglich ist.
Für $\hat{q} \in \mathrm{Q} - \{\bar{q}\}$ muß es, da F zusammenhängend ist, nach Definition 2.18 eine eingehende Kante $\delta(\hat{q}', \hat{f}) = \hat{q}$ geben. Da $\delta_f = \delta \cup \delta_{\mathrm{Q}}$, gibt es diese Kante auch in F_f, also gilt $\delta_f(\hat{q}', \hat{f}) = \hat{q}$. Aus der Eigenschaft (c) aus Definition 2.29 für h'_f folgt, daß $\delta'_f(\mathrm{h}'_f(\hat{q}'), \hat{f}) = \mathrm{h}'_f(\hat{q})$. Da nach Annahme $\mathrm{h}'_f(\hat{q}) = \bar{q}'_f$, besagt dies, daß $\bar{q}'_f$ eine eingehende Kante besitzt. Da durch die Konstruktion in Definition 2.24 nur die Kante $\delta'_{\mathrm{Q}}(\bar{q}'_f, f) = \bar{q}'$ neu eingeführt wird, muß die eingehende Kante in δ' liegen, also $\delta'(\mathrm{h}'_f(\hat{q}'), \hat{f}) = \bar{q}'_f$. Durch den Definitionsbereich von δ' folgt, daß $\bar{q}'_f \in \mathrm{Q}'$. Aus $\bar{q}'_f = \mathrm{Q}'$ folgt $\mathrm{Q}' \in \mathrm{Q}'$ und damit auch für diesen Fall ein Widerspruch.
Damit ist Behauptung 2.14 gezeigt. Die gesuchte Funktion h' kann nun konstruiert werden als $\forall q \in \mathrm{Q} : \mathrm{h}'(q) := \mathrm{h}'_f(q)$. Für h' müssen wiederum die Eigenschaften (a) bis (c) aus Definition 2.29 gezeigt werden.

Für Eigenschaft (a) ist zu zeigen, daß $\mathrm{h}'(\bar{q}) = \bar{q}'$. Es gilt

$$\begin{array}{rrcll}
\delta_f(\bar{q}_f, f) = \bar{q} \Rightarrow & \delta'_f(\mathrm{h}'_f(\bar{q}_f), f) & = & \mathrm{h}'_f(\bar{q}) & \text{(Eigenschaft 2.29 (c) für } \mathrm{h}'_f) \\
\Rightarrow & \delta'_f(\bar{q}'_f, f) & = & \mathrm{h}'_f(\bar{q}) & \text{(Eigenschaft 2.29 (a) für } \mathrm{h}'_f) \\
\Rightarrow & \bar{q}' & = & \mathrm{h}'_f(\bar{q}) & \text{(nach Definition von } \delta'_f) \\
\Rightarrow & \bar{q}' & = & \mathrm{h}'(\bar{q}) & \text{(nach Definition von h}').
\end{array}$$

Für Eigenschaft (b) ist zu zeigen, daß $\forall q \in \mathrm{Q} : \theta(q) \sqsubseteq \theta'(\mathrm{h}'(q))$.
Es gilt für alle $q \in \mathrm{Q}$, daß

$$\begin{array}{rrcll}
 & \theta_f(q) & \sqsubseteq & \theta'_f(\mathrm{h}'_f(q)) & \text{(Eigenschaft 2.29 (b) für } \mathrm{h}'_f) \\
\Rightarrow & \theta(q) & \sqsubseteq & \theta'_f(\mathrm{h}'_f(q)) & \text{(nach Definition von } \theta_f) \\
\Rightarrow & \theta(q) & \sqsubseteq & \theta'_f(\mathrm{h}'(q)) & \text{(nach Definition von h}') \\
\Rightarrow & \theta(q) & \sqsubseteq & \theta'(\mathrm{h}'(q)) & \text{(nach Definition von } \theta'_f).
\end{array}$$

Für Eigenschaft (c) ist zu zeigen, daß $f' \in \mathrm{Feat}, q_1, q_2 \in \mathrm{Q} : \delta(q_1, f') = q_2 \Rightarrow \delta'(\mathrm{h}'(q_1), f') = \mathrm{h}'(q_2)$.
Es gilt für alle $f' \in \mathrm{Feat}, q_1, q_2 \in \mathrm{Q}$:

$$\begin{array}{rrcll}
 & \delta_f(q_1, f') = q_2 & \Rightarrow & \delta'_f(\mathrm{h}'_f(q_1), f') = \mathrm{h}'_f(q_2) & \text{(Eigenschaft 2.29 (c) für } \mathrm{h}'_f) \\
\Rightarrow & \delta(q_1, f') = q_2 & \Rightarrow & \delta'_f(\mathrm{h}'_f(q_1), f') = \mathrm{h}'_f(q_2) & \text{(Def. von } \delta_f) \\
\Rightarrow & \delta(q_1, f') = q_2 & \Rightarrow & \delta'_f(\mathrm{h}'(q_1), f') = \mathrm{h}'(q_2) & \text{(Def. von h}') \\
\Rightarrow & \delta(q_1, f') = q_2 & \Rightarrow & \delta'(\mathrm{h}'(q_1), f') = \mathrm{h}'(q_2) & \text{(Def. von } \delta'_f).
\end{array}$$

Damit ist die Behauptung gezeigt, daß $F_f \sqsubseteq F'_f \Rightarrow F \sqsubseteq F'$ und somit der Beweis beendet. □

Die Subsumption stellt eine transitive und reflexive, aber nicht antisymmetrische Relation über Feature Structures dar, weil es möglich ist, daß sich verschiedene Feature Structures gegenseitig subsumieren. Damit ist die Subsumption keine partielle Ordnung über Feature Structures.

Mithilfe der Subsumption kann beschreiben werden, welche Feature Structures als vom Informationsgehalt her gleichwertig angesehen werden. Dies sind gerade sogenannte alphabetische Varianten, die sich gegenseitig subsumieren.

Definition 2.32 [Alphabetische Varianten von Feature Structures]
Zwei zusammenhängende Feature Structures F, F' über einem Typsystem TS heißen genau dann *alphabetische Varianten*, $F \sim F'$, wenn gilt:

$$F \sqsubseteq F' \wedge F' \sqsubseteq F \qquad \Diamond$$

Carpenter zeigt, daß der Begriff „alphabetische Variante“ berechtigt ist, da sich Feature Structures F und F', welche die in der Definition genannten Bedingungen erfüllen, tatsächlich nur in der Benennung ihrer Knoten unterscheiden können, ansonsten aber Strukturgleich sind. Dies entspricht dem Begriff der Isomorphie bei

Graphen oder insbesondere Moore-Automaten (siehe [Conway 1971]). Diese Eigenschaft der alphabetischen Varianten wird durch folgenden Satz ausgedrückt.

Satz 2.33 [alphabetische Varianz ist Äquivalenzrelation]
Die alphabetische Varianz $\sim$ ist eine Äquivalenzrelation über der Menge der Feature Structures. □

Oben wurde gesagt, daß die Subsumption *keine* partielle Ordnung über Feature Structures darstellt. Mithilfe der alphabetischen Varianz läßt sich der folgende Satz formulieren.

Satz 2.34 [Feature-Structure-Subsumption ist partielle Ordnung modulo alphabetische Varianten]
Die Feature-Structure-Subsumptionsrelation $\sqsubseteq$ ist eine partielle Ordnung über der Menge der Feature Structures $\mathrm{FS}_{\mathrm{TS}}$ modulo alphabetische Varianten. □

Die letzten drei Sätze werden in [Carpenter 1992] bewiesen.

Der Begriff der alphabetischen Varianten ist hilfreich für die Definition der Unifikation von Feature Structures, die wie bei Typen zu zwei konsistenten Feature Structures eine eindeutige kleinste obere Schranke liefern sollte. Diese Eindeutigkeit wird bei Feature Structures modulo alphabetische Varianz erzielt. Der folgende Abschnitt widmet sich der Unifikation von Feature Structures im Detail.

Unifikation von Feature Structures

Wie für die Subsumption gibt es eine entsprechende Erweiterung der Join-Bildung auf Feature Structures. Diese nennt man *Unifikation*. Die Unifikation zweier Feature Structures ergibt die allgemeinste Feature Structure, die alle zu unifizierenden subsumiert. Da der Join nicht zwischen allen Typen definiert sein muß, kann auch die Unifikation zweier Feature Structures undefiniert sein, was als Scheitern der Unifikation bezeichnet wird.

Carpenter konstruiert in seiner Definition eine Äquivalenzrelation, die dazu dient, Knoten zu identifizieren. Dabei steht $[x]_{\bowtie}$ für die Äquivalenzklasse, zu der x bezüglich $\bowtie$ gehört und $\mathrm{M}/_{\bowtie}$ für die Menge aller Äquivalenzklassen, in die $\bowtie$ die Menge M zerlegt. Die neuen Knoten sind damit Mengen über Knoten der beteiligten Feature Structures.

Definition 2.35 [Untypisierte Feature-Structure-Unifikation]
Seien $F, F' \in \mathrm{FS}_{\mathrm{TS}}$ Feature Structures, so daß $F \sim \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle$ und $F' \sim \langle \mathrm{Q}', \bar{q}', \theta', \delta' \rangle$ so beschaffen sind, daß $\mathrm{Q} \cap \mathrm{Q}' = \emptyset$. Wir definieren eine Relation $\bowtie$ über $\mathrm{Q} \cup \mathrm{Q}'$ als die kleinste Äquivalenzrelation, so daß

(a) $\bar{q} \bowtie \bar{q}'$ und

(b) $\forall q \in \mathrm{Q}, q' \in \mathrm{Q}', f \in \mathrm{Feat}: \quad q \bowtie q' \wedge (\delta(q,f) \text{ und } \delta'(q',f) \text{ sind definiert}) \Rightarrow \delta(q,f) \bowtie \delta'(q',f).$

Die untypisierte (genauer: nicht-wohltypisierte) Unifikation von F und F', $F \sqcup_{\mathrm{U}} F'$, sei dann definiert als

$$\begin{array}{rrcl} & F \sqcup_{\mathrm{U}} F' & := & \langle (\mathrm{Q} \cup \mathrm{Q}')/_{\bowtie}, [\bar{q}]_{\bowtie}, \theta^{\bowtie}, \delta^{\bowtie} \rangle \\ \text{mit} & \forall q \in \mathrm{Q} \cup \mathrm{Q}' : \theta^{\bowtie}([q]_{\bowtie}) & := & \bigsqcup\{(\theta \cup \theta')(q') \mid q' \bowtie q\} \\ \text{und} & \forall q \in \mathrm{Q} \cup \mathrm{Q}', f \in \mathrm{Feat}^* : \delta^{\bowtie}([q]_{\bowtie}, f) & := & \begin{cases} [(\delta \cup \delta')(q,f)]_{\bowtie} & \text{wenn } (\delta \cup \delta')(q,f) \text{ def.} \\ \text{undefiniert} & \text{sonst} \end{cases} \end{array}$$

wenn alle Joins in der Definition von $\theta^{\bowtie}$ existieren. $F \sqcup_{\mathrm{U}} F'$ ist sonst undefiniert. $\Diamond$

Daß diese Definition überhaupt zu einer wohlgeformten Feature Structure führt, welche die für Unifikation erwünschten Eigenschaften aufweist, wird in [Carpenter 1992] bewiesen. Es wird gezeigt, daß die Unifikation zweier Feature Structures $F \sqcup_{\mathrm{U}} F'$ zu einer Feature Structure führt, die (wie der Join auf Typen) die kleinste obere Schranke von $\{F, F'\}$ mit der Subsumptionsrelation als Ordnung auf Feature Structures modulo alphabetische Varianten darstellt.

Wie für Typen, kann man auch die Unifikation über Mengen von Feature Structures definieren. Dafür wird folgende Notation benutzt:

$$\bigsqcup_{\mathrm{U}}\{F_1, F_2, \ldots, F_n\} := F_1 \sqcup_{\mathrm{U}} F_2 \sqcup_{\mathrm{U}} \ldots \sqcup_{\mathrm{U}} F_n$$

Durch die Eigenschaften der kleinsten oberen Schranke führt wegen der Assoziativität der untypisierten Unifikation jede Reihenfolge der Unifikationen auf der rechten Seite zu demselben Ergebnis, die Unifikation einer Menge von Feature Structures ist also eindeutig modulo alphabetische Varianten.

Unifikation wird in dieser Arbeit häufig auch in Kombination mit dem Vorpfad-Operator eingesetzt, um mehrere Feature Structures disjunkt zusammenzufassen. Durch das Umbenennen der Knoten der an der Unifikation beteiligten Feature Structures werden diese gewissermaßen kopiert. Der folgende Satz 2.36 ähnelt Satz 2.27, in dem ein Vorpfad-Operator und dann ein Pfadwert mit jeweils demselben Pfad auf eine Feature Structure angewandt werden, was in derselben Feature Structure resultiert. Der Unterschied besteht darin, daß im folgenden Satz erlaubt wird, daß eine andere Feature Structure, die den Pfad *nicht* enthält, hinzu unifiziert wird. Es stellt sich heraus, daß die durch die Unifikation mit der anderen Feature Structure hinzugefügte Information durch den Pfadwert wieder vollständig entfernt wird. Durch mögliche Umbenennung der Knoten ergibt sich allerdings die etwas schwächere Aussage, daß eine alphabetische Variante entsteht. Weiterhin wird nicht ein Pfad, sondern ein einzelnes Feature betrachtet. Der Satz läßt sich mit Sicherheit auf Pfade verallgemeinern; da er in dieser Arbeit aber nur für Features benötigt wird, soll die folgende Version ausreichen.

Satz 2.36 [Neutrale Unifikation]
Sei $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ ein Typsystem, $f \in Feat$ ein Feature und seien $F, F' \in \mathrm{CFS}_{\mathrm{TS}}$ Feature Structures, wobei F' keine vom Wurzelknoten abgehende

Kante mit dem Feature f enthält. Dann gilt

$$(f{::}F \sqcup_{\mathrm{U}} F')@f \sim F$$

Beweis Wir zeigen die Behauptung, indem gezeigt wird, daß

$$F \sqsubseteq (f{::}F \sqcup_{\mathrm{U}} F')@f \text{ und } (f{::}F \sqcup_{\mathrm{U}} F')@f \sqsubseteq F.$$

Aus Satz 2.34 folgt, daß für beliebige Feature Structures G und G' gilt: $G \sqsubseteq G \sqcup_{\mathrm{U}} G'$. Es gilt demnach

$$\begin{array}{rcll} f{::}F & \sqsubseteq & f{::}F \sqcup_{\mathrm{U}} F' & \\ \Rightarrow (f{::}F)@f & \sqsubseteq & (f{::}F \sqcup_{\mathrm{U}} F')@f & \text{(nach Satz 2.31)} \\ \Rightarrow F & \sqsubseteq & (f{::}F \sqcup_{\mathrm{U}} F')@f & \text{(nach Satz 2.27)} \end{array}$$

Es bleibt zu zeigen, daß $(f{::}F \sqcup_{\mathrm{U}} F')@f \sqsubseteq F$.

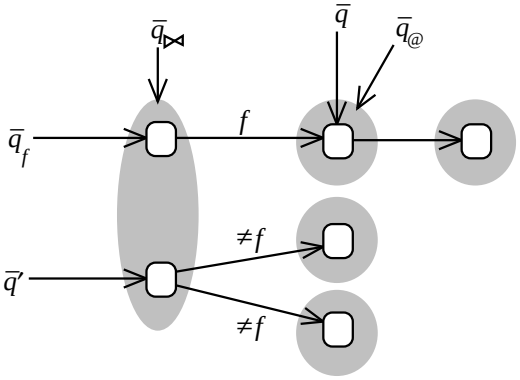

Abbildung 2.17: Skizze zum Beweis von Satz 2.36.

Sei $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle$ und $F' = \langle \mathrm{Q}', \bar{q}', \theta', \delta' \rangle$. Nach der Annahme aus Satz 2.36 gilt $\delta'(\bar{q}', f)$ ist undefiniert. Sei $F_f = \langle \mathrm{Q}_f, \bar{q}_f, \theta_f, \delta_f \rangle := f{::}F$. Aus Definition 2.24 ergibt sich $\bar{q}_f = Q\ \mathrm{Q}_f := \mathrm{Q} \cup \{\bar{q}_f\}$, sowie $\theta_f(\bar{q}_f) := \top$, $\forall q \in \mathrm{Q} : \theta_f(q) := \theta(q)$ und $\delta_f(\bar{q}_f, f) = \bar{q}$, $\forall q \in \mathrm{Q}, f' \in \mathrm{Feat}^* : \delta_f(q, f') := \delta(q, f')$.
Sei $\bowtie$ die Äquivalenzrelation aus Definition 2.35 für die Unifikation

$$F_f \sqcup_{\mathrm{U}} F' =: F^{\bowtie} = \langle \mathrm{Q}^{\bowtie}, \bar{q}^{\bowtie}, \theta^{\bowtie}, \delta^{\bowtie} \rangle.$$

Wir nehmen o.B.d.A. an, daß die oben für F_f und F' angegebenen Knotenmengen bereits disjunkt sind, also $\mathrm{Q}_f \cap \mathrm{Q}' = \emptyset$. Die Relation $\bowtie$ wird über der Menge $\mathrm{Q}_f \cup \mathrm{Q}'$ definiert. Aus der Konstruktion von $\bowtie$ folgt aus Definition 2.35 (a), daß $\bar{q}_f \bowtie \bar{q}'$. Da

$\delta_f(\bar{q}_f, f')$ nur für $f' = f$ definiert ist, $\delta(\bar{q}', f)$ nicht definiert ist und in Definition 2.35 die kleinste Äquivalenzrelation verlangt wird, werden durch Definition 2.35 (b) keine weiteren Knoten identifiziert. Die übrigen Knoten aus $\mathrm{Q}_f \cup \mathrm{Q}'$ ergeben sich als $\bar{\mathrm{Q}} := \mathrm{Q}_f \cup \mathrm{Q}' - \{\bar{q}_f, \bar{q}\} = (\mathrm{Q}_f \bar{q}_f) \cup \mathrm{Q}' - \{\bar{q}\} = \mathrm{Q} \cup \mathrm{Q}' - \{\bar{q}\}$. Für diese übrigen Knoten gilt

$$\forall q \in \bar{\mathrm{Q}} : [q]_{\bowtie} = \{q\}.$$

Aus Definition 2.35 sich, daß $\bar{q}^{\bowtie} = [\bar{q}_f]_{\bowtie} = [\bar{q}']_{\bowtie} = \{\bar{q}_f, \bar{q}'\}$ und $\mathrm{Q}^{\bowtie} := (\mathrm{Q}_f \cup \mathrm{Q}')/_{\bowtie} = \{\bar{q}^{\bowtie}\} \cup \{\{q\} \mid q \in \bar{\mathrm{Q}}\}$.
Sei weiter wie in Definition 2.35 $\forall q \in \mathrm{Q}_f \cup \mathrm{Q}' : \theta^{\bowtie}([q]_{\bowtie}) := \bigsqcup\{(\theta_f \cup \theta')(q') \mid q' \bowtie q\}$. Für den neuen Wurzelknoten ergibt sich der Typ $\theta^{\bowtie}(\bar{q}^{\bowtie}) = \theta^{\bowtie}([\bar{q}_f]_{\bowtie}) = \bigsqcup\{(\theta_f \cup \theta')(\bar{q}_f), (\theta_f \cup \theta')(\bar{q}')\} = \bigsqcup\{\theta_f(\bar{q}_f), \theta'(\bar{q}')\} = \bigsqcup\{\top, \theta'(\bar{q}')\} = \theta'(\bar{q}')$.
Sei $q \in \mathrm{Q}$, dann gilt $\theta_f(q) = \theta(q)$ und damit

$$\begin{aligned}
\theta^{\bowtie}([q]_{\bowtie}) &= \bigsqcup\{(\theta_f \cup \theta')(q') \mid q' \bowtie q\} \\
&= \bigsqcup\{(\theta_f \cup \theta')(q') \mid q' \in [q]_{\bowtie}\} \\
&= \bigsqcup\{(\theta_f \cup \theta')(q') \mid q' \in \{q\}\} \\
&= \bigsqcup\{(\theta_f \cup \theta')(q)\} \\
&= (\theta_f \cup \theta')(q) \\
&= \theta_f(q) \\
&= \theta(q).
\end{aligned}$$

Die Kantenfunktion ergibt sich nach Definition 2.35 zu

$$\forall q \in \mathrm{Q}_f \cup \mathrm{Q}', f' \in \mathrm{Feat}^* : \delta^{\bowtie}([q]_{\bowtie}, f') := \begin{cases} [(\delta_f \cup \delta')(q, f')]_{\bowtie} & \text{wenn } (\delta_f \cup \delta') \text{ def.} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Durch Betrachtung weiterer Fallunterscheidungen nach Definition A.2 und nach Definition von δ_f, nach der $\bar{q}_f \xrightarrow[\delta_f]{f} \bar{q}$ die einzige neue Kante ist, erhält man

$$\forall q \in \mathrm{Q}_f \cup \mathrm{Q}', f' \in \mathrm{Feat}^* : \delta^{\bowtie}([q]_{\bowtie}, f') := \begin{cases} [\bar{q}]_{\bowtie} & \text{wenn } q \in \{\bar{q}_f, \bar{q}'\} \wedge f' = f \\ [\delta(q, f')]_{\bowtie} & \text{wenn } q \in \mathrm{Q} \\ [\delta'(q, f')]_{\bowtie} & \text{wenn } q \in \mathrm{Q}' \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Sei $F_@ = \langle \mathrm{Q}_@, \bar{q}_@, \theta_@, \delta_@ \rangle := F^{\bowtie}@f$. Es bleibt zu zeigen, daß $F_@ \sqsubseteq F$. Wir zeigen dies wie zuvor durch die Definition einer Funktion $\mathrm{h} : \mathrm{Q}_@ \to \mathrm{Q}$, welche die Eigenschaften (a) bis (c) aus Definition 2.29 erfüllt.
Zunächst zeigen wir, daß $\mathrm{Q}_@ = \{\{\mathrm{q}\} \mid \mathrm{q} \in \mathrm{Q}\}$. Nach Definition 2.19 gilt $\mathrm{Q}_@ \subseteq \mathrm{Q}^{\bowtie}$. Es ist also zu zeigen, daß $[q]_{\bowtie} \in \mathrm{Q}_@ \iff q \in \mathrm{Q}$.

Aus der Konstruktion in Definition 2.19 folgt, daß

$$\begin{aligned}\bar{q}_@ &= \delta^{\bowtie}(\bar{q}^{\bowtie}, f)\\ &= \delta^{\bowtie}([\bar{q}_f]_{\bowtie}, f)\\ &= [(\delta_f \cup \delta')(\bar{q}_f, f)]_{\bowtie}\\ &= [\delta_f(\bar{q}_f, f)]_{\bowtie}\\ &= [\bar{q}]_{\bowtie}\\ &= \{\bar{q}\}.\end{aligned}$$

Aus der Konstruktion des Pfadwerts (Definition 2.19) folgt, daß

$$\begin{aligned}\mathrm{Q}_@ &= \{\delta^{\bowtie}(\bar{q}_@, \pi) \mid \pi \in \mathrm{Feat}^*\}\\ &= \{\delta^{\bowtie}([\bar{q}]_{\bowtie}, \pi) \mid \pi \in \mathrm{Feat}^*\}\end{aligned}$$

Dies besagt, daß $\mathrm{Q}_@$ aus genau den Knoten $[q]_{\bowtie} \in \mathrm{Q}^{\bowtie}$ besteht, für die es einen Pfad $\pi = f_1, \ldots, f_n$ gibt mit $[\bar{q}]_{\bowtie} \xrightarrow{f_1}_{\delta^{\bowtie}} [q_1]_{\bowtie} \cdots [q_{n-1}]_{\bowtie} \xrightarrow{f_n}_{\delta^{\bowtie}} [q_n]_{\bowtie} = [q]_{\bowtie}$. Aus $\bar{q} \in \mathrm{Q}$ und $\delta^{\bowtie}([\bar{q}]_{\bowtie}, f_1) = [q_1]_{\bowtie}$ folgt nach Definition von $\delta^{\bowtie}$, daß $[q_1]_{\bowtie} = [\delta(\bar{q}, f_1)]_{\bowtie}$. Da $\delta(\bar{q}, f_1) \in \mathrm{Q}$ folgt, daß $[q_1]_{\bowtie} = \{\delta(\bar{q}, f_1)\}$ und damit $q_1 = \delta(\bar{q}, f_1) \in \mathrm{Q}$. Per Induktion über n folgt $\forall i \in \{0, \ldots, n\} : q_i \in \mathrm{Q}$ und damit $q_n = q \in \mathrm{Q}$. Da $\mathrm{Q}_@$ aus genau den Knoten $[q]_{\bowtie}$ besteht und F zusammenhängend ist, folgt die Behauptung, daß $[q]_{\bowtie} \in \mathrm{Q}_@ \Leftrightarrow q \in \mathrm{Q}$ und damit $\mathrm{Q}_@ = \{\{\mathrm{q}\} \mid \mathrm{q} \in \mathrm{Q}\}$.
Für den entsprechend eingeschränkten Urbildbereich ist $\theta_@ = \theta^{\bowtie}$ und $\delta_@ = \delta^{\bowtie}$.
Sei $\forall q \in \mathrm{Q} : \mathrm{h}(\{q\}) := q$.
Eigenschaft (a) gilt, da $\mathrm{h}(\bar{q}_@) = \mathrm{h}(\{\bar{q}\}) = \bar{q}$.
Eigenschaft (b) gilt ebenfalls, da $\forall \{q\} \in \mathrm{Q}_@ : \theta_@(\{q\}) = \theta^{\bowtie}(\{q\}) = (\theta \cup \theta')(q) = \theta(q) = \theta(\mathrm{h}(\{q\}))$.
Für Eigenschaft (c) ist zu zeigen, daß $\forall f' \in \mathrm{Feat}, \{q_1\}, \{q_2\} \in \mathrm{Q}_@ : \delta_@(\{q_1\}, f') = \{q_2\} \Rightarrow \delta(\mathrm{h}(\{q_1\}), f') = \mathrm{h}(\{q_2\})$. Es ist $\delta_@(\{q_1\}, f') = \delta^{\bowtie}(\{q_1\}, f') = [\delta_f(q_1, f')]_{\bowtie} = [\delta(q_1, f')]_{\bowtie} = \{\delta(q_1, f')\}$ und damit $q_2 = \delta(q_1, f')$. Dann ist $\delta(\mathrm{h}(\{q_1\}), f') = \delta(q_1, f') = q_2 = \mathrm{h}(\{q_2\})$. □

Neben Feature Structures und Pfaden wird in der Literatur auch oft der Begriff der *Pfadgleichung* (*path equation*, [Carpenter und Thomason 1990]) verwendet. Eine Pfadgleichung sagt aus, daß in einer Feature Structure über zwei Pfade derselbe Knoten erreicht wird. Eine Pfadgleichung wird in dieser Arbeit als spezielle Feature Structure eingeführt, die durch zwei Pfade eindeutig modulo alphabetische Varianten bestimmt wird.

Definition 2.37 [Pfadgleichung]
Sei $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ ein Typsystem und seien $\pi, \pi' \in \mathrm{Feat}^*$ Pfade. Eine *Pfadgleichung* $\pi \doteq \pi'$ sei eine allgemeinste Feature Structure $F \in \mathrm{FS}_{\mathrm{TS}}$, für die gilt:

$$F@\pi = F@\pi' \qquad \Diamond$$

Es soll gezeigt werden, wie eine solche Feature Structure effektiv gefunden werden kann. Im folgenden Beweis der Konstruktion wird deutlich, daß es zu zwei Pfaden tatsächlich *eine* allgemeinste Pfadgleichung (modulo alphabetische Varianten) gibt. Es wird wie bei der Definition des Vorpfads eine Konstruktion gewählt, die eindeutige Knotenobjekte liefert und dabei die für Pfadgleichungen erwünschte Eigenschaft der Symmetrie berücksichtigt. Die Pfadgleichung wird nach dem Prinzip der Unifikation in Definition 2.35 aus einer Äquivalenzrelation über einer Knotenmenge konstruiert.

Definition 2.38 [Konstruktion einer Pfadgleichung]
Sei $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ ein Typsystem. Der *Pfadgleichungskonstruktor* ist eine Funktion $\mathrm{patheq} : (\mathrm{Feat}^*) \times (\mathrm{Feat}^*) \to \mathrm{CFS}_{\mathrm{TS}}$.
Seien $\pi, \pi' \in \mathrm{Feat}^*$ Pfade und sei Π die Menge aller Präfixe von π oder π', also

$$\Pi := \{\hat{\pi} \in \mathrm{Feat}^* \mid \exists \tilde{\pi} \in \mathrm{Feat}^* : (\hat{\pi}\,\tilde{\pi} = \pi \vee \hat{\pi}\,\tilde{\pi} = \pi')\}.$$

Sei $\equiv$ die kleinste Äquivalenzrelation über Π, für die gilt $\pi \equiv \pi'$. Dann ergibt sich patheq als

$$\begin{array}{lrcl} & \mathrm{patheq}(\pi, \pi') & := & \langle \Pi/_{\equiv}, [\varepsilon]_{\equiv}, \theta^{\equiv}, \delta^{\equiv} \rangle \\ \text{mit} & \forall \tilde{\pi} \in \Pi : \theta^{\equiv}([\tilde{\pi}]_{\equiv}) & := & \top \\ \text{und} & \forall \tilde{\pi}\, f \in \Pi : \delta^{\equiv}([\tilde{\pi}]_{\equiv}, f) & := & [\tilde{\pi}\, f]_{\equiv}. \end{array} \qquad \Diamond$$

In dieser Konstruktion werden Mengen über Präfixen von π und π' als Knotenbezeichner benutzt. Zwei Pfade stehen genau dann in der Relation $\equiv$, wenn sie gleich sind (da es sich um eine Äquivalenzrelation handelt) oder wenn es sich um die gleichzusetzenden Pfade π und π' handelt.

Für die Definition der Kantenfunktion $\delta^{\equiv}$ muß gezeigt werden, daß jedem Argumentepaar $([\tilde{\pi}]_{\equiv}, f)$ für $\tilde{\pi}\, f \in \Pi$ höchstens ein Funktionswert zugewiesen wird. Dafür zeigen wir, daß aus $\tilde{\pi}\, f \in \Pi$ und $\tilde{\pi}'\, f \in \Pi$ sowie $[\tilde{\pi}]_{\equiv} = [\tilde{\pi}']_{\equiv}$ folgt, daß $\tilde{\pi} = \tilde{\pi}'$.

Aus $[\tilde{\pi}]_{\equiv} = [\tilde{\pi}']_{\equiv}$ folgt $\tilde{\pi} \equiv \tilde{\pi}'$, woraus nach Definition der Äquivalenzrelation $\equiv$ folgt, daß $\tilde{\pi} = \tilde{\pi}'$ oder $\{\tilde{\pi}, \tilde{\pi}'\} = \{\pi, \pi'\}$. Im ersten Fall ist die Behauptung gezeigt. Nehmen wir für den zweiten Fall o.B.d.A. an, daß gilt $\tilde{\pi} = \pi$ und $\tilde{\pi}' = \pi'$. Aus $\tilde{\pi}\, f \in \Pi$ folgt, daß $\tilde{\pi}\, f$ und damit $\pi\, f$ ein Präfix von π oder π' sein muß. Da $\pi\, f$ kein Präfix von π sei kann, muß $\pi\, f$ ein Präfix von π' sein, also muß π ein *echter* Präfix von π' sein. Analog folgt für $\tilde{\pi}'$ und π', daß π' ein echter Präfix von π sein muß. Da es keine Pfade geben kann, die echte Präfixe voneinander sind, kommt dieser Fall nicht in Frage. Damit ist gezeigt, daß für alle existierenden Fällt gilt, daß $\tilde{\pi} = \tilde{\pi}'$ und damit, daß $\delta^{\equiv}$ wohldefiniert ist.

Abbildung 2.18 veranschaulicht die Benennung von Knoten einer Pfadgleichung, da deren Verständnis für den folgenden Beweis wichtig ist. Sei π_1 der längste gemeinsame Präfix von π und π' und $\pi = \pi_1\, \pi_2$ sowie $\pi' = \pi_1\, \pi_2'$. Jeder Knoten ist mit der Menge der Pfade benannt, die zu ihm führen. Nur zum „Zielknoten“ $\{\pi, \pi'\}$ führen

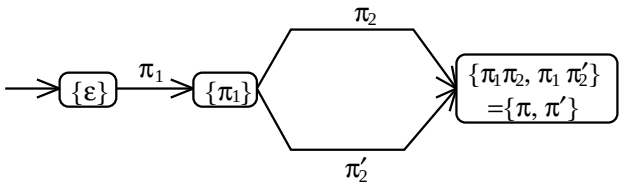

Abbildung 2.18: Skizze zur Knotenbenennung in Pfadgleichungen nach Definition 2.38.

zwei im Normalfall unterschiedliche Pfade, weshalb alle anderen Knoten durch einelementige Mengen benannt werden.

Es gibt verschiedene Fälle, in denen einige der in der Abbildung gezeigten Knoten zusammenfallen: Falls $\pi_1 = \varepsilon$ (die Pfade π und π' haben keinen gemeinsamen Präfix), fallen die Knoten $\{\varepsilon\}$ und $\{\pi_1\}$ zusammen. Falls π ein Präfix von π' ist, gilt $\pi_2 = \varepsilon$, wodurch die beiden rechten Knoten zusammenfallen und eine Schleife (zyklische Struktur) mit $\pi_2' \neq \varepsilon$ entsteht. Der umgekehrte Fall, daß π' ein Präfix von π ist, führt zu einem analogen Resultat. Falls $\pi = \pi'$, entfällt zusätzlich die Schleife. Für den Fall $\pi = \pi' = \varepsilon$ fallen alle Knoten im Knoten $\{\varepsilon\}$ zusammen und es werden keine Kanten konstruiert. Eine Pfadgleichung über zwei gleiche Pfade stellt einen Sonderfall dar, der üblicherweise nicht vorkommen wird, da eine solche Feature Structure mit $\pi::[\top]$ auch als Vorpfad konstruiert werden kann.

Satz 2.39 [patheq **erzeugt Pfadgleichungen**]
Die Funktion patheq aus Definition 2.38 erzeugt Pfadgleichungen nach Definition 2.37.

Beweis Wir gehen im folgenden von den Bezeichnern aus der Konstruktion in Definition 2.38 aus.
Nach Definition 2.37 ist eine Pfadgleichung die *allgemeinste* Feature Structure, welche die Pfadwerte unter zwei Pfaden π und π' identifiziert. Satz 2.39 behauptet, daß $P := \text{patheq}(\pi, \pi')$ eine Pfadgleichung ist. Demnach ist zu zeigen, daß $P@\pi = P@\pi'$, und daß für alle $P' \in \text{FS}_{\text{TS}}$ mit $P'@\pi = P'@\pi'$ gilt, daß $P \sqsubseteq P'$.
Wir zeigen zunächst, daß $P@\pi = P@\pi'$. Nach Definition 2.16 folgt für die Erweiterung der Funktion $\delta^{\equiv}$ aus Definition 2.38 auf Pfade:

$$\forall \tilde{\pi} \in \Pi : \delta^{\equiv}([\varepsilon]_{\equiv}, \tilde{\pi}) := [\tilde{\pi}]_{\equiv}.$$

Demnach ist $\delta^{\equiv}([\varepsilon]_{\equiv}, \pi) = [\pi]_{\equiv} = \{\pi, \pi'\} = [\pi']_{\equiv} = \delta^{\equiv}([\varepsilon]_{\equiv}, \pi')$. Nach Satz 2.22 folgt aus der Gleichheit der Knoten die Gleichheit der Pfadwerte, also gilt $P@\pi = P@\pi'$.
Es bleibt zu zeigen, daß gilt: $\forall P' \in \text{FS}_{\text{TS}} : P'@\pi = P'@\pi' \Rightarrow P \sqsubseteq P'$.
Sei $P' = \langle \text{Q}', \bar{q}', \theta', \delta' \rangle$. Wir zeigen die Subsumption, indem wir eine Funktion $\text{h} : \Pi/_{\equiv} \rightarrow \text{Q}'$ angeben, welche die Eigenschaften (a) bis (c) aus Definition 2.29 erfüllt.

Bevor wir h definieren, zeigen wir zunächst, daß aus $\bar{\pi} \equiv \bar{\pi}'$ folgt, daß $\delta'(\bar{q}', \bar{\pi}) = \delta'(\bar{q}', \bar{\pi}')$. Aufgrund der Konstruktion der Äquivalenzrelation $\equiv$ müssen genau zwei Fälle unterschieden werden: Es gilt $\bar{\pi} = \bar{\pi}'$, oder aber $\bar{\pi}$ und $\bar{\pi}'$ sind die Pfade π und π'.
Für den Fall $\bar{\pi} = \bar{\pi}'$ ist nur zu zeigen, daß $\delta'(\bar{q}', \bar{\pi})$ definiert ist. Aus $\bar{\pi} \in \Pi$ folgt, daß $\bar{\pi}$ ein Präfix von π oder π' ist. Nach Voraussetzung ist $P'@\pi = P'@\pi'$, woraus nach Definition 2.19 des Pfadwerts folgt, daß $\delta'(\bar{q}', \pi)$ und $\delta'(\bar{q}', \pi')$ definiert sind. Also muß δ' auch für alle Präfixe von π oder π' definiert sein.
Für den zweiten Fall nehmen wir o.B.d.A. an, daß $\bar{\pi} = \pi$ und $\bar{\pi}' = \pi'$. Wie oben gezeigt sind $\delta'(\bar{q}', \pi)$ und $\delta'(\bar{q}', \pi')$ definiert. Nach Definition 2.19 hat $P'@\pi$ den Wurzelknoten $\delta'(\bar{q}', \pi)$, analog hat $P'@\pi'$ den Wurzelknoten $\delta'(\bar{q}', \pi')$. Da nach Voraussetzung $P'@\pi = P'@\pi'$ gilt, sind insbesondere deren Wurzelknoten gleich, also $\delta'(\bar{q}', \pi) = \delta'(\bar{q}', \pi')$.
Unter der gezeigten Voraussetzung können wir h eindeutig definieren als

$$\forall \tilde{\pi} \in \Pi : \mathrm{h}([\tilde{\pi}]_{\equiv}) := \delta'(\bar{q}', \tilde{\pi}).$$

Für h müssen wiederum die Eigenschaften (a) bis (c) aus Definition 2.29 gezeigt werden.
Eigenschaft (a) ist gezeigt durch $\mathrm{h}([\varepsilon]_{\equiv}) = \delta'(\bar{q}', \varepsilon) = \bar{q}'$.
Eigenschaft (b) gilt, da alle Knoten in P den Typ *Top* haben, und somit für alle $\tilde{\pi} \in \Pi$ gilt $\theta^{\equiv}([\tilde{\pi}]_{\equiv}) = \top \sqsubseteq \theta'(\mathrm{h}([\tilde{\pi}]_{\equiv})) = \theta'(\delta'(\bar{q}', \tilde{\pi}))$.
Zuletzt werde Eigenschaft (c) gezeigt. Sei $[\tilde{\pi}_1]_{\equiv} \xrightarrow[\delta^{\equiv}]{f} [\tilde{\pi}_2]_{\equiv}$ eine Kante in P, dann ist zu zeigen, daß gilt $\delta'(\mathrm{h}([\tilde{\pi}_1]_{\equiv}), f) = \mathrm{h}([\tilde{\pi}_2]_{\equiv})$. Aus der Existenz der Kante in P folgt nach Definition von $\delta^{\equiv}$, daß $[\tilde{\pi}_1\, f]_{\equiv} = [\tilde{\pi}_2]_{\equiv}$ und $\tilde{\pi}_1\, f \in \Pi$. Es ist

$$\begin{array}{rcll} \delta'(\mathrm{h}([\tilde{\pi}_1]_{\equiv}), f) & = & \delta'(\delta'(\bar{q}', \tilde{\pi}_1), f) & \text{(nach Definition von h)} \\ & = & \delta'(\bar{q}', \tilde{\pi}_1\, f) & \text{(nach Definition 2.16)} \\ & = & \mathrm{h}([\tilde{\pi}_1\, f]_{\equiv}) & \text{(da } \tilde{\pi}_1\, f \in \Pi\text{)} \\ & = & \mathrm{h}([\tilde{\pi}_2]_{\equiv}). & \end{array}$$

Somit ist gezeigt, daß die Funktion patheq Pfadgleichungen erzeugt. □

Damit sind die für diese Arbeit notwendigen grundlegenden Definitionen und Sätze für Feature Structures gegeben. Im nächsten Abschnitt werden als Erweiterung wohltypisierte Feature Structures eingeführt.

2.3.4 Wohltypisierte Feature Structures

In Abschnitt 2.1.2 wurde eine Feature-Typisierung als Teil eines Typsystems eingeführt. Die bisher eingeführten Feature Structures sind zwar insofern typisiert, als jeder Knoten einen Typ zugewiesen bekommt. Die Funktion approp wurde bisher aber noch nicht verwendet, so daß die definierten Feature Structures beliebige Features mit Werten von beliebigem Typ enthalten können. Carpenter nennt Feature

Structures, die nach der Feature-Typisierung ihres Typsystems angemessene Features und Typen verwenden, *wohltypisierte* Feature Structures.

Der Typ eines Knotens ist dann angemessen, wenn dieser von allen Typen subsumiert wird, die sich aus Feature-Typisierungen von Features ergeben, die auf diesen Knoten zeigen.

Definition 2.40 [wohltypisierte Feature Structure]
Eine Feature Structure $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle$ über einem Typsystem $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ heißt *wohltypisiert*, wenn F zusammenhängend ist ($F \in \mathrm{CFS_{TS}}$) und für alle $q, q' \in \mathrm{Q}, \mathrm{f} \in \mathrm{Feat}$ gilt

$$\delta(q, f) = q' \Rightarrow \mathrm{approp}(\theta(q), f) \sqsubseteq \theta(q'). \qquad \Diamond$$

Wohltypisierte Feature Structures dürfen also für einen Knoten höchstens die Features definieren, die durch die Feature-Typisierung erlaubt sind, und die zugewiesenen Werte müssen mindestens vom verlangten Typ sein.

Sei $\mathrm{WFS_{TS}}$ die Menge aller wohltypisierten Feature Structures über dem Typsystem TS.

Der folgende Satz macht eine Aussage über den minimalen Typ, von dem ein Knoten in einer wohltypisierten Feature Structure sein muß. Jeder Knoten muß demnach als Typ einen Subtyp des Join aller der Typen erhalten, die sich aus Feature-Typisierungen von Features ergeben, die auf diesen Knoten zeigen.

Satz 2.41 [Minimaler Knotentyp in einer wohltypisierten Feature Structure]
Sei $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle$ eine Feature Structure über einem Typsystem $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$. Für die Knotentypisierungsfunktion θ gilt für alle $q, q' \in \mathrm{Q}$ und $f \in \mathrm{Feat}$:

$$\bigsqcup \{\mathrm{approp}(\theta(q), f) \mid \delta(q, f) = q'\} \sqsubseteq \theta(q')$$

Beweis Nach der Definition des Join als Supremum folgt direkt aus Definition A.9:

$$\bigsqcup \mathrm{T} \sqsubseteq \tau \Rightarrow \forall \sigma \in \mathrm{T} : \sigma \sqsubseteq \tau$$

Die zu zeigende Forderung aus Satz 2.41 kann man also umschreiben in:

$$\forall q, q' \in \mathrm{Q}, f \in \mathrm{Feat}, \sigma \in \{\mathrm{approp}(\theta(q), f) \mid \delta(q, f) = q'\} : \sigma \sqsubseteq \theta(q')$$

$$\Longrightarrow \forall q, q' \in \mathrm{Q}, f \in \mathrm{Feat} : \delta(q, f) = q' \Rightarrow \mathrm{approp}(\theta(q), f) \sqsubseteq \theta(q')$$

Dies entspricht genau der Definition von wohltypisierten Feature Structures in 2.40. □

Es stellt sich die Frage, ob die Unifikation von wohltypisierten Feature Structures wiederum zu einer wohltypisierten Feature Structure führt (falls sie nicht scheitert). Nach der Definition der Feature-Typisierung ist ein erlaubtes Feature auch in allen

Subtypen erlaubt. Da durch Unifikation ein Knotentyp nur spezieller werden kann, können im Unifikationsergebnis also keine unerlaubten Features auftreten.

Der Typ eines Feature-Werts kann allerdings von einem Subtypen eingeschränkt werden, so daß ein im Supertyp erlaubter Wert-Typ im Subtyp verboten sein kann. Hier gilt es, zwei Fälle zu unterscheiden: Entweder, der Typ ist nicht erlaubt, weil er zwar kompatibel, aber zu allgemein ist. Dieser Fall ist leicht zu korrigieren, indem das Unifikationsergebnis durch Spezialisierung nachträglich wohltypisiert gemacht wird, wozu eine *Typisierung* dient. Der andere Fall wird weiter unten vor Definition 2.45 behandelt.

Definition 2.42 [Typisierung]
Eine Typisierungsprozedur oder kurz *Typisierung* sei eine partielle Funktion $\mathrm{t} : \mathrm{FS}_{\mathrm{TS}} \hookrightarrow \mathrm{WFS}_{\mathrm{TS}}$, die eine Feature Structure $F \in \mathrm{FS}_{\mathrm{TS}}$ auf eine allgemeinste wohltypisierte Feature Structure $F' \in \mathrm{WFS}_{\mathrm{TS}}$ abbildet, für die gilt $F \sqsubseteq F'$. Eine Feature Structure F heißt mit t *wohltypisierbar*, wenn $\mathrm{t}(F)$ definiert ist. $\Diamond$

Eine Typisierung ist also eine Funktion, die zu einer (wohltypisierbaren) Feature Structure die allgemeinste wohltypisierte Feature Structure liefert, die von dieser subsumiert wird. Betrachten wir Feature Structures modulo alphabetische Varianten, so ergibt sich eine eindeutige allgemeinste Feature Structure. Die zu wohltypisierende Feature Structure wird also gerade soweit spezialisiert, bis sie wohltypisiert ist. Es sei darauf hingewiesen, daß Definition 2.42 nicht fordert, daß die Funktion t für *jede* Feature Structure definiert ist, für die es einen Funktionswert geben könnte.

Die folgende Definition konstruiert eine Funktion typify, die nach dem darauf folgenden Satz ein Typisierung ist.

Carpenter definiert zu einem ähnlichen Zweck eine Typ-Inferenzprozedur, geht aber im Gegensatz zum hier vorgestellten Ansatz von Typsystemen aus, in denen es für jedes Feature einen allgemeinsten Typ gibt, der dieses Feature einführt. Diese Einschränkung ist zwar insofern praktisch, als es damit möglich ist, komplett untypisierte Feature Structures automatisch eindeutig zu typisieren. Hier soll diese Einschränkung jedoch nicht gemacht werden, da es in der Praxis häufig vorkommt, daß Features in mehreren Typen, die nicht in einer Vererbungsbeziehung stehen, vorkommen. Weiterhin ist die komplette Typinferenz im Bereich der Objektorientierung, auf den hier Bezug genommen wird, nicht üblich. Bei der Notation von Feature Structures wird in Abschnitt 2.3.5 erlaubt, solche Typbeschriftungen wegzulassen, die sich aus der im folgenden beschriebenen Typisierungsprozedur eindeutig ergeben. Dies stellt eine minimale Form der Typinferenz dar.

Definition 2.43 [Konstruktion von typify**]**
Sei $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ ein Typsystem. Die Funktion $\mathrm{typify} : \mathrm{FS}_{\mathrm{TS}} \hookrightarrow \mathrm{WFS}_{\mathrm{TS}}$ sei definiert, indem man zu $F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle \in \mathrm{FS}_{\mathrm{TS}}$ den Funktionswert $\mathrm{typify}(F)$ nach folgendem Algorithmus konstruiere:

1. Starte mit $i := 0$ und $\theta_0 = \theta_i := \theta$.

2. Für alle $q \in \mathrm{Q}$ setze
   $\theta_{i+1}(q) := \theta_i(q) \sqcup \bigsqcup\{\mathrm{approp}(\theta_i(q'), f) \mid f \in \mathrm{Feat}, q' \in \mathrm{Q}, \delta(q', f) = q\}$
   (für jeden Knoten wird dessen Typ mit den durch die eingehenden Kanten geforderten Typen unifiziert).

3. Falls mindestens einer dieser Joins nicht definiert ist, scheitert die Typisierung und typify(F) ist undefiniert.

4. Ist $\theta_{i+1} \neq \theta_i$, setze $i := i + 1$ und gehe zu 2.

5. Ist $\theta_{i+1} = \theta_i$, setze typify(F) := $\langle \mathrm{Q}, \bar{q}, \theta_i, \delta \rangle$, wenn dies eine wohltypisierte Feature Structure ist, andernfalls ist typify(F) undefiniert. Terminiere. $\Diamond$

Intuitiv verfeinert typify die Knotentypen von F mit der durch approp erhältlichen Information. Auch wenn keine weitere Information aus approp mehr ergänzt werden kann, muß in Schritt 5 geprüft werden, ob die resultierende Typisierungsfunktion zu einer wohltypisierten Feature Structure führt, da es in F immer noch Features geben kann, die auch mit den Knotentypen aus dem letzten θ_n nicht erlaubt sind. Als Beispiel nehme man an, daß vor der Typisierung alle Knoten einen Typ tragen, der keinerlei Features erlaubt. Dann kann durch typify auch kein Typ verfeinert werden und die resultierende Feature Structure ist nicht wohltypisiert, obwohl alle Joins definiert waren. typify(F) ist dann undefiniert, obwohl es bei einem entsprechenden Typsystem eine Knotentypisierungsfunktion geben könnte, die F wohltypisiert. Wir werden später sehen, daß typify jedoch ausreicht, um das Ergebnis einer Unifikation aus wohltypisierten Feature Structures genau dann wohltypisiert zu machen, wenn eine wohltypisierte speziellere Feature Structure existiert.

Es muß noch gezeigt werden, daß der gegebene Algorithmus terminiert. Da man durch Join-Bildung in Schritt 2 stets einen mindestens so speziellen Typ erhält, gilt als Schleifeninvariante für alle $i \in \{1 \ldots n\}$ und $q \in \mathrm{Q} : \theta_{\mathrm{i}}(\mathrm{q}) \sqsubseteq \theta_{\mathrm{i+1}}(\mathrm{q})$. Da die Menge der Typen endlich ist und ein Typ damit nur endlich oft spezialisiert werden kann, muß nach einer endlichen Anzahl n die Abbruchsbedingung $\theta_{i+1} = \theta_i$ gelten.

Satz 2.44 [typify **ist eine Typisierung**]
Die in Definition 2.43 definierte Funktion typify ist eine Typisierung nach Definition 2.42.

Beweis Es muß nur der Fall betrachtet werden, in dem typify(F) definiert ist, da Definition 2.42 keine Aussage über den anderen Fall macht.
Zu zeigen ist, daß typify(F) eine allgemeinste wohltypisierte Feature Structure ist, die von F subsumiert wird. Daß typify(F) wohltypisiert ist, wird explizit in Schritt 5 des Algorithmus in Definition 2.43 gefordert. Es bleibt also zu zeigen, daß $F \sqsubseteq$ typify(F) und daß typify(F) die allgemeinste wohltypisierte Feature Structure ist, welche diese Bedingung erfüllt. Wir zeigen dies durch Induktion über i. Für jedes $F_i = \langle \mathrm{Q}, \bar{q}, \theta_i, \delta \rangle$ ist zu zeigen, daß für alle wohltypisierten $F' \in \mathrm{WFS}_{\mathrm{TS}}$ gilt: $F \sqsubseteq$

$F' \Rightarrow F_i \sqsubseteq F'$. In der Konstruktion werden den F_i weder Knoten noch Kanten hinzugefügt, noch werden Knoten identifiziert, um zu typify(F) zu gelangen. Ein F', das allgemeiner wäre als ein F_i, müßte also auch dieselbe Knotenmenge und dieselben Kanten aufweisen, da sich sonst leicht ein allgemeineres finden ließe. Wir zeigen deshalb die Behauptung o.B.d.A. nur für alle wohltypisierten $F' = \langle \mathrm{Q}, \bar{q}, \theta', \delta \rangle \in \mathrm{WFS}_{\mathrm{TS}}$ mit $F \sqsubseteq F'$. Zu zeigen ist für alle $i \in \{0 \ldots n\} : F_i \sqsubseteq F'$.
Für $i = 0$ gilt $F_0 = F$. Damit folgt aus $F \sqsubseteq F'$ trivialerweise $F_0 \sqsubseteq F'$.
Sei die Behauptung für alle $i \in \{0, \ldots, k\}$ bewiesen, dann gilt insbesondere $F_k \sqsubseteq F'$, und damit:

$$\forall q \in \mathrm{Q} : \theta_k(q) \sqsubseteq \theta'(q) \tag{2.19}$$

Zu zeigen ist nun, daß $F_{k+1} \sqsubseteq F'$ gilt. Dazu ist eine entsprechende Subsumptionsabbildung h aus Definition 2.29 zu finden, die sich in diesem Fall als die Identitätsabbildung $\mathrm{h} : \mathrm{Q} \to \mathrm{Q}$ ergibt. Weil F_{k+1} denselben Wurzelknoten $\bar{q}$ und dieselbe Kantenfunktion δ benutzt, ist nur noch zu zeigen, daß die Bedingung 2.29 (c) für θ_{k+1} gilt:

$$\forall q \in \mathrm{Q} : \theta_{k+1}(q) \sqsubseteq \theta'(q) \tag{2.20}$$

Seien im folgenden $f \in \mathrm{Feat}$ und $\hat{q} \in \mathrm{Q}$. Die Induktionsregel lautet nach Schritt 2 aus dem Algorithmus:

$$\forall q \in \mathrm{Q} : \theta_{k+1}(q) = \theta_k(q) \sqcup \bigsqcup \{\mathrm{approp}(\theta_k(\hat{q}), f) \mid \delta(\hat{q}, f) = q\} \tag{2.21}$$

Nach Definition 2.3 gilt $\sigma \sqsubseteq \tau \Rightarrow \mathrm{approp}(\sigma, f) \sqsubseteq \mathrm{approp}(\tau, f)$. Mit (2.19) folgt $\mathrm{approp}(\theta_k(\hat{q}), f) \sqsubseteq \mathrm{approp}(\theta'(\hat{q}), f)$, und damit

$$\forall q \in \mathrm{Q} : \theta_{k+1}(q) \sqsubseteq \theta_k(q) \sqcup \bigsqcup \{\mathrm{approp}(\theta'(\hat{q}), f) \mid \delta(\hat{q}, f) = q\} \tag{2.22}$$

Da F' wohltypisiert ist, gilt nach Satz 2.41:

$$\forall q \in \mathrm{Q} : \bigsqcup \{\mathrm{approp}(\theta'(\hat{q}), f) \mid \delta(\hat{q}, f) = q\} \sqsubseteq \theta'(q) \tag{2.23}$$

Wegen $\sigma \sqsubseteq \tau \wedge \upsilon \sqsubseteq \tau \Rightarrow \sigma \sqcup \upsilon \sqsubseteq \tau$ erhält man aus (2.23) zusammen mit (2.19):

$$\forall q \in \mathrm{Q} : \theta_k(q) \sqcup \bigsqcup \{\mathrm{approp}(\theta'(\hat{q}), f) \mid \delta(\hat{q}, f) = q\} \sqsubseteq \theta'(q) \tag{2.24}$$

Durch Verkettung der Subsumptionen aus (2.22) und (2.24) erhält man genau die Behauptung (2.20). □

Die Typisierung wurde oben motiviert als eine Funktion, die das Ergebnis einer Unifikation aus wohltypisierten Feature Structures so spezialisieren kann, daß es ebenso wohltypisiert ist. Dabei wurde der Fall, in dem das Unifikationsergebnis nicht wohltypisierbar ist, noch nicht betrachtet.

Ein Beispiel für ein solches nicht-wohltypisierbares Unifikationsergebnis zeigt Abbildung 2.19. Die Typen A1 und A2 sind disjunkte Subtypen von A, der ein Feature

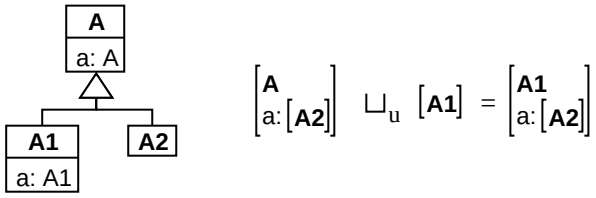

Abbildung 2.19: Eine Unifikation von wohltypisierten Feature Structures, deren Ergebnis nicht wohltypisierbar ist.

a vom Typ A definiert. Im Subtyp A1 wird dieses Feature auf A1 spezialisiert. Bei beiden Feature Structures links vom Gleichheitszeichen sind wohltypisiert. Das Ergebnis der untypisierten Unifikation ist definiert, es kann aber keine Typisierung geben, welche für das Unifikationsergebnis definiert ist, da A1 und A2 inkompatibel sind.

Eine *wohltypisierte Unifikation* sollte in so einem Fall scheitern.

Definition 2.45 [wohltypisierte Unifikation]
Die *wohltypisierte Unifikation* einer endlichen Menge von Feature Structures $\mathbb{F} = \{F_1, F_2, \ldots, F_n\}$ sei definiert als die Typisierung ihrer (untypisierten) Unifikation mit typify, sofern diese definiert ist:

$$\bigsqcup \mathbb{F} := \begin{cases} \text{typify}(\bigsqcup_{\text{U}} \mathbb{F}) & \text{falls } \bigsqcup_{\text{U}} \mathbb{F} \text{ und typify}(\bigsqcup_{\text{U}} \mathbb{F}) \text{ definiert sind} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Für die Unifikation zweielementiger Mengen benutzt man auch Infixnotation:

$$F \sqcup F' := \bigsqcup \{F, F'\} \qquad \Diamond$$

Eine erwünschte Eigenschaft, die noch zu zeigen wäre, ist, daß die wohltypisierte Unifikation von wohltypisierten Feature Structures nur dann scheitert, wenn es *keine* Typisierungsfunktion t gibt, die für das Ergebnis der untypisierten Unifikation definiert ist. Wir behaupten, daß durch die Verwendung der Typisierungsfunktion typify in der Definition der wohltypisierten Unifikation diese Eigenschaft gilt. Bewiesen wird diese Behauptung hier nicht, da in dieser Arbeit wohltypisierte Unifikation nur dort eingesetzt wird, wo das Ergebnis auch ohne Anwendung von typify wohltypisiert oder undefiniert ist.

Im weiteren wird der Zusatz „wohltypisierte“ Unifikation weggelassen, stattdessen wird explizit auf untypisierte Unifikation von Feature Structures hingewiesen.

Man beachte, daß die wohltypisierte Unifikation nicht voraussetzt, daß die beteiligten Feature Structures wohltypisiert oder auch nur wohltypisierbar sind. Dies hat folgenden Grund: Pfadgleichungen (Definition 2.37) und auch der Vorpfad-Operator

(Definition 2.24) erzeugen im allgemeinen keine wohltypisierten Feature Structures, da den neuen Knoten der Typ $\top$ zugewiesen wird. $\top$ erlaubt normalerweise aber keinerlei Features. Die resultierende Feature Structure ist meist nicht einmal mit typify wohltypisierbar, da bei der Typisierung beispielsweise der Typ von Knoten, die von keinem anderen Knoten referenziert werden, nicht spezialisiert werden kann. Für Pfadgleichungen und Vorpfade (bei nicht-leerem Pfad) gilt dies unter anderem für den Wurzelknoten.

Pfadgleichungen und der Vorpfad-Operator werden aber meist so verwendet, daß die untypisierten (genauer: mit $\top$ typisierten) Knoten mit typisierten Knoten unifiziert werden, so daß die untypisierte Unifikation einer Feature Structure mit einer als Pfadgleichung oder über den Vorpfad-Operator erzeugten Feature Structure durchaus wohltypisiert oder zumindest wohltypisierbar sein kann. Ein Beispiel hierfür mit dem Typsystem aus Abbildung 2.19 ist die folgende wohltypisierte Unifikation:

$$[\mathsf{A1}] \sqcup \mathsf{a}::[\mathsf{A}] = [\mathsf{A1}] \sqcup \begin{bmatrix} \top \\ \mathsf{a}: [\mathsf{A}] \end{bmatrix} = \text{typify}([\mathsf{A1}] \sqcup_{\mathrm{U}} \begin{bmatrix} \top \\ \mathsf{a}: [\mathsf{A}] \end{bmatrix}) = \text{typify}(\begin{bmatrix} \mathsf{A1} \\ \mathsf{a}: [\mathsf{A}] \end{bmatrix}) = \begin{bmatrix} \mathsf{A1} \\ \mathsf{a}: [\mathsf{A1}] \end{bmatrix}$$

Nachdem die formalen Grundlagen von Feature Structures eingeführt bzw. erarbeitet wurden, werden im folgenden Abschnitt einige hilfreiche Kurznotationen und Tupel und Listen als spezielle standardisierte Typen eingeführt.

2.3.5 Erweiterte Notationen für Feature Structures

Mit Feature Structures können komplexe Datenobjekte basierend auf einem Typsystem beschrieben und als Graphen oder AVMs dargestellt werden. Zur kompakteren Darstellung unter anderem von Standard-Datenstrukturen ist es nützlich, zusätzlichen „syntaktischen Zucker“ einzuführen. In diesem Abschnitt werden einige allgemeine Abkürzungen sowie spezielle Notationen für die häufig benutzten Datentypen Tupel und Liste eingeführt, wobei speziell für Listen eine typisierte Variante angegeben wird. Anschließend wird eine UML-artige Notation für Feature Structures dargestellt.

Allgemeine Kurznotationen

Als erste Vereinfachung sei erlaubt, den Typ $\top$ (*top*) wegzulassen, also die Feature Structure $[\top]$ zu notieren als $[]$.

In Abschnitt 2.3.4 wurde eine Funktion typify eingeführt, die in der Lage ist, bestimmte nicht-wohltypisierte Feature Structures in wohltypisierte umzuwandeln. Wenn klar ist, daß eine typisierte Feature Structure spezifiziert werden soll, sei erlaubt, daß eine nicht-wohltypisierte Feature Structure F notiert wird, für die typify(F) definiert ist. Damit spart man sich in vielen Fällen die explizite Typisierung von Feature-Werten, wenn diese aus dem Typsystem hervorgeht. Es wird dennoch empfohlen, den Typ eines Feature-Werts zu notieren, wenn dieser sich nicht

trivial aus dem Typsystem ergibt. Dies ist beispielsweise der Fall, wenn Knotenreferenzen dazu führen, daß zwei durch approp gegebene Typen unifiziert werden müssen, um zu dem für den Feature-Wert erlaubten Typ zu gelangen.

Tupel

Ein *Tupel* besteht aus einer Anzahl von Komponenten, wobei die Anzahl als *Arität* bezeichnet wird. Der direkte Zugriff auf die i-te Komponente (i liegt zwischen 1 und der Arität des Tupels) soll bei einem Tupel möglichst einfach sein.

Ein Tupel läßt sich als Feature Structure darstellen, indem im zugrundeliegenden Typsystem für jede gewünschte Arität n ein Typ Tupel$n$ und ein Feature cn definiert wird. Jeder Typ Tupel$n$ erlaubt genau die Features c1, c2, ..., cn und erlaubt jeweils den Typ $\top$ für jedes Feature ci (approp(Tupel$n$, ci) $= \top$ für alle $i \in \{1, \ldots, n\}$, undefiniert sonst). Wichtig ist, daß die gewünschten Tupeltypen vor Konstruktion des Typsystems bekannt sind, damit die Menge der Typen endlich bleibt.

Da der Typ $\top$ in der Regel keinerlei Features erlauben sollte, liegt es nahe, Tupel0 $= \top$ zu setzen.

Es wäre weiterhin auch denkbar, einen Typ Tupel einzuführen, der alle Typen Tupel$n$ subsumiert, worauf hier aber verzichtet werden soll. Eine Verwendung von Tupeln unbekannter Arität erscheint nicht sinnvoll, da die Arität gerade die einzige Information ist, die ein Tupel zur Verfügung stellt. Insofern enthielte der Tupel-Typ keinerlei Information. Für eine Kollektion unbekannter Länge sollte vielmehr der im folgenden eingeführte Listen-Typ verwendet werden. Aus demselben Grund werden auch die verschiedenen Tupel-Typen nicht in eine Subsumptionsbeziehung gesetzt. Dies ist ein gutes Beispiel für den Unterschied zwischen Vererbung und Ist-Ein-Beziehung: Die Wiederverwendung der Features von Tupel$n-1$ in Tupel$n$ legt eine Vererbungsbeziehung nahe, doch ein n-Tupel ist kein $n-1$-Tupel, sondern es handelt sich um nicht-unifizierbare (inkompatible) Typen.

Da die Features einer Tupel-Feature-Structure durchnumeriert sind, sollten sie in einer kompakten Darstellung weggelassen werden können. Wir führen als „syntaktischen Zucker“ für die AVM-Notation ein, daß für einen Knoten vom Typ Tuple$n$ der Typ und die Feature-Bezeichner entfallen können. Statt der so entstehenden vertikalen Anordnung können alternativ die Komponenten horizontal angeordnet werden. Man rekonstruiert die Tupel-Feature-Structure, indem man dem Feature ci, $i \in \{1 \ldots n\}$ gerade den i-ten Knoten zuweist. Hier kommt es also im Gegensatz zur AVM-Notation auf die Reihenfolge der Unterstrukturen an, die in der vertikalen Notation von oben nach unten, in der horizontalen von links nach rechts gelesen wird.

Abbildung 2.20 zeigt ein Tupel in ausführlicher Darstellung (links) und in abkürzender Schreibweise (rechts).

Damit der Typ Tuple$n$ eindeutig aus der Notation hervorgeht, muß der Wert n eindeutig erkennbar sein, also wird gefordert, daß alle n Komponenten aufgezählt

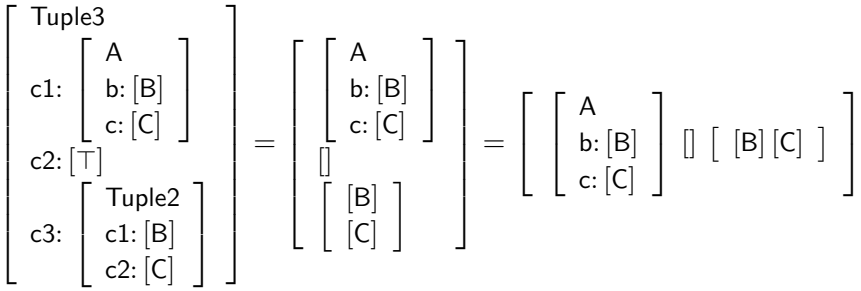

Abbildung 2.20: Ein verschachteltes Tupel als Feature Structure in ausführlicher (links) und abkürzender Notation (rechts). Komponenten können vertikal oder horizontal angeordnet werden.

werden. Sollen Komponenten unterspezifiziert bleiben, so notiert man für die i-te Komponente einfach die Feature Structure []. Nur für Null-Tupel erhält man mit dieser Notation eine scheinbar mehrdeutige Notation, da für [Tupel0] die Notation [] erlaubt ist. Durch die oben geforderte Gleichheit Tuple0 $= \top$ ergibt sich aber für beide Interpretation von [] dieselbe Feature Structure, nämlich [] = [Tuple0] = [⊤] = [].

Listen

Eine weitere häufig verwendete Datenstruktur, die in dieser Arbeit für viele Beispiele herangezogen wird, ist die Liste. Eine nicht-leere Liste wird typischerweise als ein Paar aus dem Kopf-Element der Liste (*head*) und der Rest-Liste (*tail*) dargestellt. Wir wollen hier nicht auf die Tupel-Definition von oben zurückgreifen, sondern eine nicht-leere Liste als einen Typ NEList mit den Features hd und tl modellieren. Weiterhin kann man mit Hilfe von Typ-Vererbung die Unterscheidung von leeren und nicht-leeren Listen elegant definieren: NEList und der Typ für die leere Liste, EList, werden als Subtypen eines (abstrakten) Typs List vereinbart. Das resultierende Typsystem $\mathrm{TS_L} = \langle \mathrm{Type_L}, \sqsubseteq_\mathrm{L}, \mathrm{Feat_L}, \mathrm{approp_L} \rangle$ besteht also aus folgenden Komponenten:

(a) $\mathrm{Type_L} := \{\top, \mathsf{List}, \mathsf{NEList}, \mathsf{EList}\}$

(b) $\sqsubseteq_\mathrm{L}$ sei die reflexive, transitive Hülle von $\{(\top, \mathsf{List}), (\mathsf{List}, \mathsf{NEList}), (\mathsf{List}, \mathsf{EList})\}$

(c) $\mathrm{Feat_L} := \{\mathsf{hd}, \mathsf{tl}\}$

(d) $\mathrm{approp_L}(\mathsf{NEList}, \mathsf{hd}) = \top$, $\mathrm{approp_L}(\mathsf{NEList}, \mathsf{tl}) = \mathsf{List}$

Eine alternative Bezeichnung für die Listen-Typen wird in Anlehnung an reguläre Ausdrücke gewählt. Für den abstrakten Listen-Typ List sei auch der Stern ($*$), für

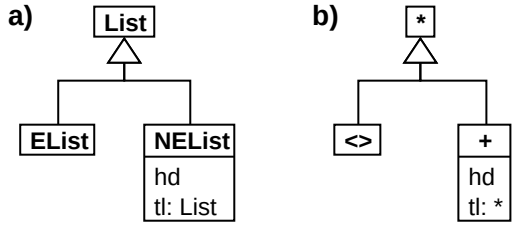

Abbildung 2.21: Das Listen-Typsystem Type_L mit ausführlichen Bezeichnern in (a) und Kurzbezeichnern in (b).

die nicht-leere Liste NEList das Pluszeichen (+) und für die leere Liste ein Kleiner-als- gefolgt von einem Größer-als-Zeichen (<>) erlaubt. Abbildung 2.21 zeigt das Listen-Typsystem Type_L mit ausführlichen Bezeichnern in (a) und Kurzbezeichnern in (b).

Seien A und B weitere Typen, so zeigt Abbildung 2.22 die Liste mit den Elementen [A...] und [B] als Feature Structure mit ausführlichen und Kurzbezeichnern sowohl als AVM (links) als auch in Graphnotation (rechts).

Da Listen sehr häufig vorkommen und auf diese Weise dargestellt recht unleserlich und platzverbrauchend erscheinen, soll auch für die AVM-Darstellung von Listen zusätzlicher „syntaktischer Zucker“ eingeführt werden.

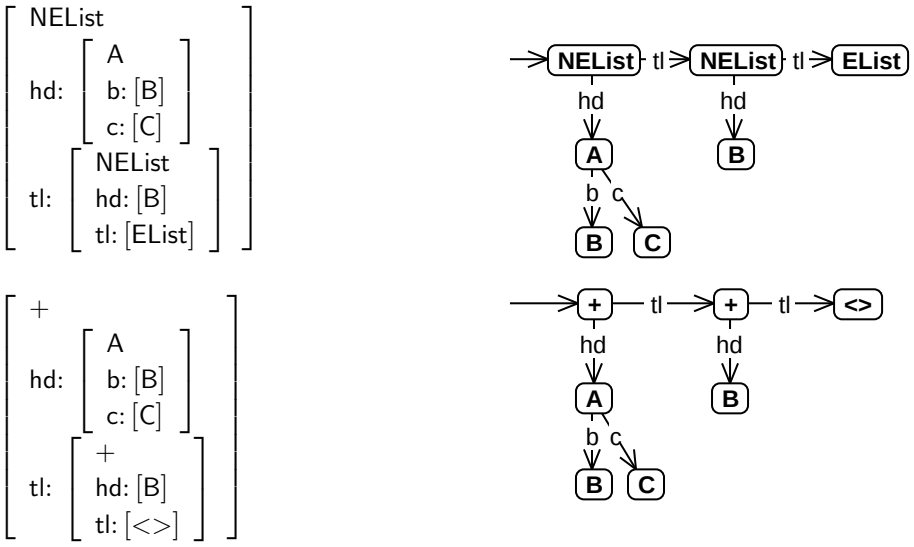

Abbildung 2.22: Eine Liste als Feature Structure mit ausführlichen und Kurzbezeichnern in AVM- und in Graphnotation.

Während eckige Klammern in einer AVM für einen Knoten der Feature Structure mit dem angegebenen Typ stehen, sollen spitze Klammern einen Knoten vom Typ EList beschreiben und spitze Klammern mit einem senkrechten Strich einen Knoten vom Typ NEList, wobei der Wert von hd vor dem senkrechten Strich und der Wert von tl hinter dem senkrechten Strich notiert werden. Abbildung 2.23 zeigt die Liste von oben in dieser modifizierten Notation. Als weitere Abkürzung kann die Rest-Liste ohne den senkrechten Strich und die spitzen Klammern notiert werden, so daß man die zweite Darstellung in Abbildung 2.23 erhält.

Man beachte, daß es mit dieser Darstellung möglich ist, Listen unbekannter Länge („offene“ Listen) darzustellen, indem man den senkrechten Strich mit einer Rest-Liste [List] kombiniert (Abbildung 2.24). Nutzt man zusätzlich die oben eingeführte Möglichkeit, den Typ List per Typinferenz zu erhalten, kann eine solche offene Liste sehr kompakt wie in Abbildung 2.24 auf der rechten Seite notiert werden.

$$\left\langle \begin{bmatrix} \mathsf{A} \\ \mathsf{b}\colon [\mathsf{B}] \\ \mathsf{c}\colon [\mathsf{C}] \end{bmatrix} \;\middle|\; \langle [\mathsf{B}] \mid \langle\rangle \rangle \right\rangle = \left\langle \begin{bmatrix} \mathsf{A} \\ \mathsf{b}\colon [\mathsf{B}] \\ \mathsf{c}\colon [\mathsf{C}] \end{bmatrix} [\mathsf{B}] \right\rangle$$

Abbildung 2.23: Die Liste aus Abbildung 2.22 als AVM in abkürzender Notation.

$$\left\langle \begin{bmatrix} \mathsf{A} \\ \mathsf{b}\colon [\mathsf{B}] \\ \mathsf{c}\colon [\mathsf{C}] \end{bmatrix} [\mathsf{B}] \;\middle|\; [\mathsf{List}] \right\rangle = \left\langle \begin{bmatrix} \mathsf{A} \\ \mathsf{b}\colon [\mathsf{B}] \\ \mathsf{c}\colon [\mathsf{C}] \end{bmatrix} [\mathsf{B}] \;\middle|\; [] \right\rangle$$

Abbildung 2.24: Eine offene Liste als AVM in abkürzender Notation.

Die Form der Listenrepräsentation wurde in Anlehnung an Prolog gewählt (siehe z.B. [Sterling und Shapiro 1994]), nicht zuletzt, da sie in FSNets auch für eine spezielle Form des *pattern matching* verwendet wird.

Typisierte Listen

Bisher wurde davon ausgegangen, daß Tupelkomponenten und Listenelemente einen beliebigen Typ haben dürfen. Ausdrucksfähiger ist die Definition von Behälterklassen wie Tupeln, Listen und Mengen als *parametrisierte Typen*. Die Parametrisierung von Typen durch Typvariablen erhöht aber gerade im Fall von Typen mit Vererbungshierarchien die Komplexität der Typtheorie sehr. Die Typmenge ist nicht mehr endlich und bereits beim Join von Typen muß ein *occurs check* durchgeführt werden, um Endlosschleifen zu vermeiden. Aus diesem Grund soll hier auf ein allgemeines Konzept parametrisierter Typen verzichtet werden. Da Listen jedoch in dieser Arbeit sehr häufig benutzt werden, beispielsweise zur Darstellung mehrwertiger Assoziationen, wird als Spezialfall erlaubt, den Typ von Listenelementen auf einfache Weise

einschränken zu können. Wir bezeichnen diese einfache Form parametrisierter Listen als typisierte Listen.

Listen benötigen genau einen Typparameter, den *Basistyp* der Liste. Jedes Element der Liste muß von diesem Basistyp oder einem seiner Subtypen sein. Da Listen wie oben eingeführt durch drei verschiedene Typen repräsentiert werden, muß für jeden dieser drei Typen ein Basistyp angegeben werden. Wir notieren den Basistyp in Klammern hinter dem Listentyp, also beispielsweise für einen Basistyp A als $\mathsf{List(A)}$, $\mathsf{NEList(A)}$ und $\mathsf{EList(A)}$. Es mag verwundern, daß auch die leere Liste nach ihrem Basistyp unterschieden werden soll, doch dies ist für eine saubere Typmodellierung unumgänglich. Alternativ können auch die Kurzbezeichner eingesetzt werden, wobei diese wiederum in Anlehnung an reguläre Ausdrücke ohne Klammern hinter dem bzw. um den Basistyp notiert werden. Die oben genannten Typen können also auch als A^*, A^+ und $<\mathsf{A}>$ geschrieben werden. Man beachte insbesondere den Unterschied zwischen der Feature Structure $\langle[\mathsf{A}]\rangle$, welche eine Liste mit einem Element vom Typ A darstellt, und dem leeren Listentyp $<\mathsf{A}>$ zum Basistyp A.

Damit solche Listentypen mit dem bisher verwendeten Typsystem dargestellt werden können, muß der Typ $\mathsf{List(A)}$ als *abkürzende Schreibweise* verstanden werden. Es gibt also nicht einen allgemeinen Typ typeList(x), der beliebig parametrisierbar ist, sondern wie bei Tupeln (siehe oben) werden die gewünschten Listenkonzepte statisch konstruiert. Damit bleibt die Typmenge endlich.

Betrachten wir nun, wie sich die Subsumptionsrelation vom Basistyp auf typisierte Listen erweitern läßt. Abbildung 2.25 zeigt in (a) eine einfache Typhierarchie, in der A und B disjunkte Subtypen des allgemeinsten Typs $\top$ sind, wobei A zwei weitere kompatible Subtypen $\mathsf{A1}$ und $\mathsf{A2}$ besitzt. In (b) und (c) sind zwei Sichten auf die resultierenden typisierten Listentypen gezeigt. Wie man sieht überträgt sich sowohl die Subsumption der Basistypen (in (b) gezeigt) als auch die Hierarchie der Listentypen aus Abbildung 2.21 (in (c) gezeigt) auf die typisierten Listentypen.

Eine Liste ist genau dann mit einer anderen Liste kompatibel, wenn ihre Listentypen *und* ihre Basistypen kompatibel sind. Als Ergebnis des Join ergibt sich der Join der Listentypen mit dem Join der Basistypen als neuen Basistyp. Da die Menge der Basistypen und die Menge der Listentypen endlich ist, bleibt durch diese Konstruktion die Typmenge endlich.

Beispiele für den Join von Listentypen sind $\mathsf{A}^* \sqcup <\mathsf{A}> = <\mathsf{A}>$ sowie $\mathsf{A1}+ \sqcup \mathsf{A2}+ = \{\mathsf{A1}, \mathsf{A2}\}+$ und $\mathsf{A}^+ \sqcup \mathsf{A1}* = \mathsf{A1}+$. Der Join scheitert beispielsweise für $<\mathsf{A}> \sqcup \mathsf{A}^+$ sowie $\mathsf{B}^* \sqcup \mathsf{A}^*$ und $\mathsf{A}^+ \sqcup \mathsf{B}^*$.

Nachdem Kurznotationen für Tupel und Listen betrachtet wurden, wird im folgenden Abschnitt eine UML-artige Notation für Feature Structures eingeführt.

Feature Structures in UML-Notation

In UML gibt es spezielle Notationen für Klassen (Abschnitt 2.1.1), Objekte (Abschnitt 2.2.1) und Objektsichten (Abschnitt 2.2.2). Feature Structures werden in

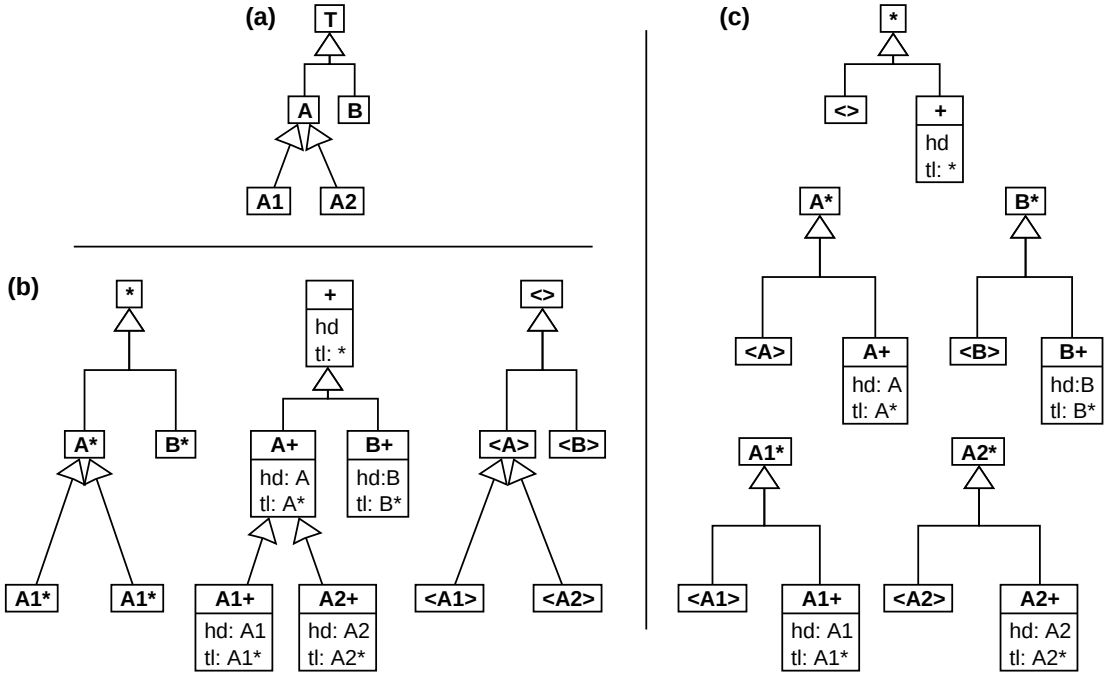

Abbildung 2.25: Erweiterung der Subsumptionsrelation von Basistypen in (a) auf Listentypen in (b) und (c).

dieser Arbeit benutzt, um Datenobjekte zu beschreiben. Insofern liegt es nahe, für Modellierer, die UML-Notationen gewohnt sind, eine Schreibweise für Feature Structures einzuführen, die sich an diesen Standard anlehnt.

Im Prinzip ist eine Feature Structure einem UML-Objekt sehr ähnlich: Beide verfügen über Attribute bzw. Features, die auf andere Objekte verweisen. Es ist aber zu beachten, daß Feature Structures nicht Objekte bzw. Objektgraphen *sind*, sondern *Bedingungen* spezifizieren, welche der konkrete Objektgraph einhalten muß. Damit ähneln sie den in Abschnitt 2.2.2 beschriebenen Objektsichten.

Entsprechend notieren wir eine Feature Structure in der UML-artigen Notation wie ein UML-Objekt, nur daß wie bei Objektsichten die Unterstreichung des Objektnamens und -typs entfällt. Damit soll verdeutlicht werden, daß es sich bei Feature Structures nicht um konkrete Klassenexemplare handelt. Features können entweder als Aggregate oder als Assoziationen dargestellt werden (siehe Abschnitt 2.2.1). Durch diese Auswahl ergeben sich ähnliche Möglichkeiten wie bei der in Abschnitt 2.3.1 eingeführten Mischnotation aus AVM- und Graphnotation. Hier wird kein neues Beispiel für eine Feature Structure in UML-artiger Notation gegeben, da beispielsweise Abbildung 2.10 ohne Unterstreichung der Objekte und mit einem entsprechend aus Abbildung 2.4 abgeleiteten Typsystem als Beispiel herangezogen

werden kann.

2.4 Implementierung einer Feature-Structure-Bibliothek

In diesem Abschnitt wird ein Überblick über die Implementierung einer Feature-Structure-Bibliothek beschrieben, die im Rahmen dieser Arbeit durchgeführt wurde. Als Programmiersprache wurde Java gewählt, da die Feature-Structure-Bibliothek wie in Abschnitt 4.5 beschrieben zur Erstellung eines FSNet-Werkzeugs als Erweiterung des Referenznetz-Werkzeugs Renew eingesetzt wird. Da Renew in Java implementiert wurde, ist die Wahl der Programmiersprache naheliegend.

Da es sich bei Java um eine objektorientierte Sprache handelt, bietet sich für Analyse und Entwurf der Feature-Structure-Bibliothek UML an. Entsprechend der Formalisierung von Feature Structures als Graphen in Abschnitt 2.3.2 ergibt sich ein objektorientiertes Grundmodell von Typen, Konzepten und Feature Structures, das in Abschnitt 2.4.1 beschrieben wird. Abschnitt 2.4.2 behandelt eine praxisorientierte Erweiterung der Implementierung, die Integration von Java-Klassen und Java-Objekten in Feature Structures. In Abschnitt 2.4.3 wird der Algorithmus zur Unifikation von Feature Structures als ein zentraler Bestandteil der Feature-Structure-Bibliothek genauer vorgestellt.

2.4.1 Objektorientiertes Grundmodell

Um eine Feature-Structure-Bibliothek objektorientiert zu implementieren, wird als erstes eine UML-Analyse-Klassendiagramm hergeleitet, welches die Definition von Konzept- und Typsystemen aus den Abschnitten 2.1.2 und 2.1.3 sowie die Definition von Feature Structures in Abschnitt 2.3.2 widerspiegelt. Aus dem Analyse-Klassendiagramm ergibt sich ein Entwurfsdiagramm, aus dem direkt die zu implementierenden Klassen abgelesen werden können.

Abbildung 2.26 zeigt das verwendete Analyse-Klassendiagramm. Zunächst sollen einige Erläuterungen zur UML-Notation gegeben werden. Für die Grundlagen sei auf Abschnitt 2.1.1 verwiesen.

Die kleinen schwarzen Dreiecke an den Assoziationen geben die *Leserichtung* für die Assoziationsnamen an. Da diese Richtung im vorliegenden Beispiel mit der Navigationsrichtung übereinstimmt, hätte die Angabe entfallen können.

Die Kästen links von den Klassen Node und Concept mit der Beschriftung feature stellen *qualifizierte Assoziationen* (*qualified associations*, siehe [Hitz und Kappel 1999]) dar. Das angegebene *qualifizierende Attribut* gehört zur Assoziation und bestimmt das Objekt am anderen Assoziationsende eindeutig. In dem weiter unten beschriebenen Entwurfsdiagrammen werden die qualifizierten Assoziationen wie allgemein üblich nicht verwendet, sondern durch entsprechende Operationen ersetzt.

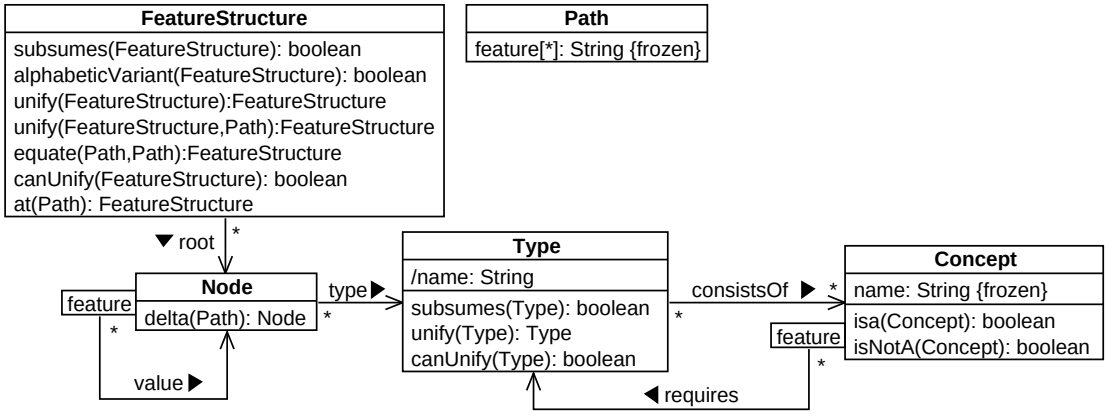

Abbildung 2.26: Analyse-Klassendiagramm des Feature-Structure-Systems.

Alle Attribute und Methoden werden wie in Analysediagrammen üblich ohne Sichtbarkeiten angegeben. Es handelt sich bei allen Klassen um ≪class≫-Stereotype.

Die Klasse FeatureStructure repräsentiert eine gesamte Feature Structure als Objekt. Der Graph setzt sich aus einzelnen Node-Objekten zusammen, wobei das FeatureStructure-Objekt auf den Wurzelknoten verweist (Assoziation root).

Die Multiplizität * an der Assoziation root besagt, daß mehrere FeatureStructure-Objekte auf denselben Wurzelknoten verweisen dürfen. Dies ist nur dann sinnvoll, wenn wie hier eine Feature Structure als *nicht-modifizierbare* Datenstruktur repräsentiert wird. Die Möglichkeit, Knoten wiederverwenden zu können, wird in der Implementierung genutzt, um den Speicherbedarf zu verringern. So liefert beispielsweise die Methode at() für den leeren Pfad die Feature Structure selbst zurück. Wollte man Modifikation von FeatureStructure-Objekten erlauben, müßte eine Kopie zurückgeliefert werden, um Alias-Probleme zu vermeiden.

Knoten haben einen Typ (Assoziation type zu Klasse Type) und verweisen über Features auf andere Knoten (Assoziation value). Ein Knoten darf von mehreren anderen Knoten referenziert werden (Multiplizität * an value). Er darf auch von demselben anderen Knoten mehrfach referenziert werden, durch die Qualifizierung der Assoziation mit feature ist dies aber nur für unterschiedliche Werte von feature erlaubt. Dies ist eine syntaktisch erwünschte Eigenschaft für Feature Structures, die auf diese Weise im Klassendiagramm dargestellt werden kann.

Die Klasse Type repräsentiert Typen, die wie in Abschnitt 2.1.3 aus mehreren Konzepten bestehen (Assoziation consistsOf zu Klasse Concept). Konzepte werden durch die Klasse Concept dargestellt, welche den Namen des Konzept als Attribut name enthält. Da der Name eines Konzepts nicht nachträglich geändert werden soll, ist dieser als {frozen} angegeben. Auch die Klasse Type hat ein Attribut name. Da sich der Name eines Typs aus den Namen seiner Konzepte ergibt, wird das Attibut

hier als abgeleitet dargestellt. Die Konzept-Feature-Typisierung aus Abschnitt 2.1.2 wird wiederum durch eine mit feature qualifizierte Assoziation repräsentiert. Einem Konzept kann über die Assoziation requires ein pro Feature eindeutiger Typ zugewiesen werden, was der Funktion approp_c aus Definition 2.9 entspricht. Analog zum Fall oben kann ein Typ von mehreren Konzepten als Feature-Typ gefordert werden (Multiplizität * an requires), aber es kann nicht dasselbe Feature im selben Konzept zwei verschiedene Typen fordern.

Schließlich dient die Klasse Path dazu, einen Pfad als eine Sequenz von Zeichenketten (mehrwertiges Attribut feature) zu repräsentieren. Diese Klasse hat zwar keine expliziten Beziehungen zu anderen Klassen, kommt aber als Parametertyp in verschiedenen Operationen vor.

Für alle Klassen werden die wichtigsten Operationen angegeben. Diese ergeben sich auf natürliche Weise aus den in den Abschnitten 2.1.2 und 2.3 eingeführten Relationen und Operatoren, wobei die Namen weitestgehend von den Funktionen übernommen wurden. Abweichend wird in der Klasse Type der Join ebenfalls als Unifikation (Operationen unify und canUnify) bezeichnet. Die Semantik der Assoziationen und Operationen aller Klassen aus Abbildung 2.26 ist zusammenfassend in Tabelle 2.1 dargestellt. Dabei werden Assoziationen durch die implizit gegebenen Navigationsmethoden spezifiziert.

In vielen Programmiersprachen, so auch in Java, gibt es die Möglichkeit, als Ergebnis einer Operation eine *Ausnahme* (*exception*) zu liefern. Dieses Konzept wird hier verwendet, wenn die gegebenen Parameter zu einem undefinierten Funktionsergebnis führen.

Der Leser mag an dieser Stelle die neu definierten Konstrukte des Vorpfad-Operators und der Pfadgleichung vermissen. Der Vorpfad ist dazu gedacht, eine Unifikation zu ermöglichen, die nicht von den Wurzelknoten ausgeht, während eine Pfadgleichung mit einer Feature Structure unifiziert wird, um (mindestens) zwei ihrer Knoten zu identifizieren. Für diese Funktionalität wurden in der Implementierung andere Operationen gewählt, die den Vorteil haben, daß es in üblichen Anwendungen keine nicht-wohltypisierten Feature Structures als Zwischenergebnisse gibt (siehe Abschnitt 2.3.4). In der formalen Darstellung lassen sich aber die oben genannten Konstrukte besser behandeln. In der Tabelle wird beschrieben, wie sich die implementierten, komplexeren Operationen durch die formal definierten Konstrukte darstellen lassen.

Das Analysediagramm zeigt die Grundstruktur der Bibliothek, aus der sich die Entwurfsklassen ergeben. Aus den formalen Definitionen aus den Abschnitten 2.1.2, 2.1.3 und 2.3 lassen sich Algorithmen zur Implementierung der Operationen durch entsprechende Methoden ableiten. In Abschnitt 2.4.3 wird als wichtigstes Beispiel der Unifikationsalgorithmus für Feature Structures beschrieben.

Abbildung 2.27 zeigt das verwendete Entwurfs-Klassendiagramm der Feature-Structure-Bibliothek. Auf Details der Abbildung wird hier nicht eingegangen. Durch das Werkzeug `javadoc` des Java *software development kit* (Java-SDK,

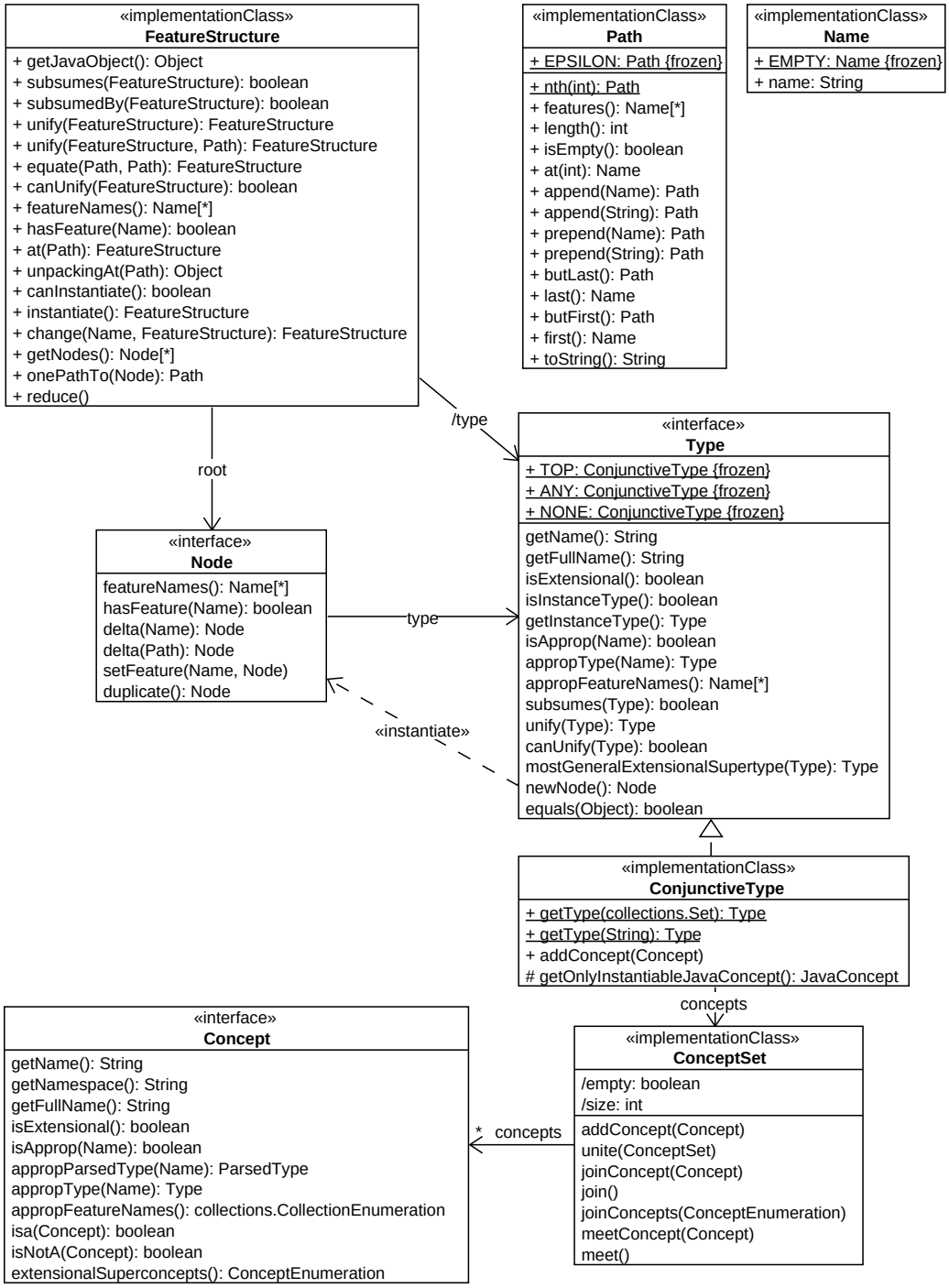

Abbildung 2.27: Entwurfs-Klassendiagramm der Feature-Structure-Bibliothek.

Operation	**Semantik**	**Ausnahme**
Klasse FeatureStructure		
$F.\mathsf{root}() = \bar{q}$	$F = \langle \mathrm{Q}, \bar{q}, \theta, \delta \rangle$	
$F.\mathsf{subsumes}(F')$	$F \sqsubseteq F'$	
$F.\mathsf{alphabeticVariant}(F')$	$F \sim F'$	
$F.\mathsf{unify}(F') = \hat{F}$	$F \sqcup F' = \hat{F}$	$F \sqcup F'$ ist undefiniert
$F.\mathsf{unify}(F', \pi) = \hat{F}$	$F \sqcup \pi{::}F' = \hat{F}$	$F \sqcup \pi{::}F'$ ist undefiniert
$F.\mathsf{equate}(\pi, \pi') = \hat{F}$	$F \sqcup \pi \doteq \pi' = \hat{F}$	$F \sqcup \pi \doteq \pi'$ ist undefiniert
$F.\mathsf{canUnify}(F')$	$F \sqcup F'$ ist definiert	
$F.\mathsf{at}(\pi) = \hat{F}$	$F@\pi = \hat{F}$	$F@\pi$ ist undefiniert
Klasse Node		
$q.\mathsf{value}(f) = q'$	$\delta(q, f) = q'$	$\delta(q, f)$ ist undefiniert
$q.\mathsf{type}() = \sigma$	$\theta(q) = \sigma$	
$q.\mathsf{delta}(\pi) = q'$	$\delta(q, \pi) = q'$	$\delta(q, \pi)$ ist undefiniert
Klasse Type		
$\sigma.\mathsf{consistsOf}() = \mathrm{C}$	$\sigma = \mathrm{C}$	
$\sigma.\mathsf{subsumes}(\tau)$	$\sigma \sqsubseteq \tau$	
$\sigma.\mathsf{unify}(\tau) = \upsilon$	$\sigma \sqcup \tau = \upsilon$	$\sigma \sqcup \tau$ ist undefiniert
$\sigma.\mathsf{canUnify}(\tau)$	$\sigma \sqcup \tau$ ist definiert	
Klasse Concept		
$c.\mathsf{requires}(f) = \sigma$	$\mathrm{approp_c}(c, f) = \sigma$	
$c.\mathsf{isa}(c')$	c ISA c'	
$c.\mathsf{isNotA}(c')$	c ISNOTA c'	

Tabelle 2.1: Zusammenhang zwischen Assoziationen und Operationen der Klassen aus Abbildung 2.26 und den formalen Definitionen aus den Abschnitten 2.1 und 2.3.

siehe [Sun 2000b]) wurde aus den Programmquellen eine detaillierte Hypertext-Dokumentation der Programmierschnittstelle (*application programming interface*, API) erstellt. Die implementierten Klassen werden mit entsprechender Dokumentation der API auf der RENEW-Website ([Kummer und Wienberg 2000]) zur Verfügung gestellt.

2.4.2 Integration von Feature Structures und Java

Die im vorherigen Abschnitt vorgestellte Feature-Structure-API ist in der Implementierung um einiges umfangreicher, als das Analysediagramm vermuten läßt. Der Grund dafür liegt in erweiterter Funktionalität, die zur Anwendung von Feature Structures benötigt wird.

In der Implementierung wurde eine nahtlose Integration von Feature Structures und Java erzielt. Hier wird ein kurzer Überblick gegeben, wie dabei vorgegangen wurde. Klassen und Schnittstellen werden in das Typsystem integriert und Java-

Objekte werden in Feature Structures erlaubt.

Integration von primitiven Typen, Klassen und Schnittstellen in das Typsystem

In Abschnitt 2.1.2 wurde davon ausgegangen, daß ein Typsystem durch eine Menge von Konzepten und Beziehungen zwischen diesen spezifiziert wird. In der Praxis gibt es weitere Anforderungen zu beachten.

- Typsysteme werden aus Bibliotheken zusammengestellt, die wiederverwendet werden sollen.
- Es werden Standard-Typen wie Zeichenketten, Zahlen, boolsche Werte etc. benötigt.
- Bestimmte Datentypen werden als Klassen zusammen mit speziellen Operationen zur Verfügung gestellt, die nutzbar sein sollen.

Alle genannten Anforderungen wurden bei der Implementierung des Typsystems berücksichtigt.

Für die erste Anforderung wird auf das Konzept der Namensräume (*name spaces*) zurückgegriffen, das auch in UML existiert ([OMG 2000]). In UML-Syntax wird der Namensraumbezeichner dem Namen des Modellelements (hier: Typ oder Klasse) mit zwei Doppelpunkten als Trennzeichen vorangestellt. Auch in Java gibt es mit den Pakten ein Namensraum-Konzept. Hier werden Paket- und Klassenname durch einen Punkt getrennt. Der Paketname kann selbst beliebig viele Punkte enthalten. Diese dienen aber nur zu einer hierarchischen Namensvergabe und etablieren keine hierarchischen Namensräume in bezug auf Sichtbarkeiten. Diese syntaktische Unterscheidung wird hier beibehalten, indem die Punkt-Notation für Java-Pakete und die Notation durch :: (nicht zu verwechseln mit dem Vorpfad-Operator aus Abschnitt 2.3.2) für Typ-Namensräume verwendet wird.

In Java können zu Beginn einer Quelldatei mit `import`-Anweisungen Klassen oder gesamte Pakete angegeben werden, die in dieser Quelldatei ohne voll qualifizierte Namen genutzt werden dürfen. Ein ähnliches Konzept wird für Typ-Namensräume erlaubt. Um auch hier eine syntaktische Trennung zu erhalten, wird zum Import von Typ-Namensräumen das Schlüsselwort `access` verwendet, das wiederum einen Bezug zu den UML-Namensräumen nahelegt. Das Stereotyp «`access`» zeigt in UML Zugriffs-Abhängigkeiten zwischen Modellelementen, beispielsweise zwischen Pakten an.

Die zweite und dritte Anforderung, die Verwendung von vordefinierten Standardtypen und der Zugriff auf bereits existierende Klassen, wurde entsprechend der Implementierungssprache Java so realisiert, daß bei der Definition von einem Typsystem auf das Java-Typsystem zurückgegriffen werden kann. Da durch diese zweite

Erweiterung Typen und Java-Klassen gemischt verwendet werden können, ist die syntaktische Unterscheidung der Namensräume besonders wichtig.

Das Java-Typsystem besteht bekanntermaßen aus zwei Teilen ([Gosling et al. 1996]): Zum einen stellt Java eine feste Anzahl primitver Typen zur Verfügung, die keine Klassen sind. Dazu gehören im einzelnen die Typen `char, byte, short, int, long, float` und `double`. Zum anderen gibt es Klassen (*classes*) und Schnittstellen (*interfaces*), die in einer Vererbungshierarchie stehen. Das Feature-Structure-System wurde nicht nur um die primitiven Typen erweitert, um der zweiten Anforderung zu genügen, sondern so gestaltet, daß auf *beliebige* Java-Typen Bezug genommen werden kann, womit die dritte Anforderung erfüllt wird.

Zu jedem primitiven Typ gibt es eine Klasse, die ein primitives Datum als Objekt kapselt (*wrapping*). Die Zeichenkette ist in Java als die Klasse `java.lang.String` realisiert. Da Zeichenketten einen sehr häufig anzutreffenden Datentyp darstellen, werden sie in Java durch eine spezielle Syntax unterstützt und sollen auch im Feature-Structure-Typsystem wie ein Basistyp behandelt werden. Der Unterschied ist für das Typsystem nicht so erheblich wie für Java selbst, da im Typsystem alle Objekte als nicht-modifizierbare Datenobjekte angesehen werden.

Java verfügt über einen Introspektionsmechanismus (Paket `java.lang.reflect`, [Gosling et al. 1996]), mit dem zur Laufzeit zu einem gegebenen Klassennamen Meta-Information abgefragt werden kann. Dieser Mechanismus erlaubt es, beliebige Java-Klassen und -Schnittstellen in das benutzerdefinierte Feature-Structure-Typsystem einzubinden. Dabei werden die Java-Klassenobjekte als *Konzepte* im Sinne der in Abschnitt 2.1.3 eingeführten Konzeptsysteme betrachtet. Java-Klassen und -Schnittstellen werden als Konzepte und nicht als Typen betrachtet, da eine konjunktive Verbindung von Klassen und einer oder mehreren Schnittstellen möglich ist. Beispielsweise kann durch einen Typ gefordert werden, daß ein Objekt vom Typ `java.awt.Point` *und* `java.lang.Cloneable` ist. Offensichtlich sind zwei Klassen aus Typ-Sicht inkompatibel, es sei denn, sie stehen in einer Subklassen-Beziehung. Durch Introspektion läßt sich diese Information über Java-Klassen gewinnen, indem die in der (Meta-)Klasse `java.lang.Class` definierte Methode `isAssignableFrom()` aufgerufen wird.

Auf die Implementierung dieser Erweiterungen des Typsystems kann hier nicht im einzelnen eingegangen werden. Entsprechende Dokumentation wird mit der nächsten Renew-Version veröffentlicht, die FSRenew enthält.

Abbildung 2.28 zeigt eine Verfeinerung der Schnittstelle Concept des Entwurfs-Klassendiagramms aus Abbildung 2.27. Hier macht sich die Trennung in eine Schnittstelle Concept und eine Standard-Implementierung ConceptImpl bezahlt, da so eine Java-Klasse als alternative Implementierung eines Konzepts als Klasse JavaConcept realisiert werden kann. Diese Klasse implementiert alle von Concept vorgegebenen Methoden und kann zusätzlich das Klassenobjekt der Klasse oder Schnittstelle liefern, welche sie repräsentiert (Methode getJavaClass()). Weiterhin kann abgefragt

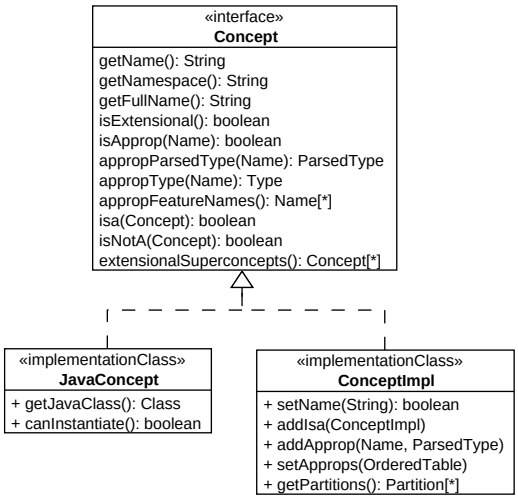

Abbildung 2.28: Entwurfs-Klassendiagramm der verschiedenen Konzeptklassen.

werden, ob Objekte dieser Java-Klasse vom Feature-Structure-System instantiierbar sind (Methode canInstantiate()). Die Bedeutung dieser Methode wird bei der Behandlung von Java-Objekten unten klar. Ein Java-Objekt kann genau dann vom Feature-Structure-System instantiiert werden, wenn es sich um eine instantiierbare Klasse (also weder eine abstrakte Klasse noch eine Schnittstelle) handelt, die einen öffentlichen Konstruktor ohne Parameter besitzt.

Integration von primitiven Daten und Java-Objekten in Feature Structures

Während sich die Integration vordefinierter *Typen* relativ leicht gestaltet, führt die Verwendung von *Daten und Objekten* zu komplexeren Entwurfsentscheidungen. In Abschnitt 2.1 wurde deutlich, daß die hier verwendeten Typen als eine Abstraktion von objektorientierten Typen aufgefaßt werden können. Feature Structures sind jedoch nicht das Gegenstück in Form der in Abschnitt 2.2 diskutierten Objekten, weil sie keine *konkreten* Objekte darstellen. Es stellt sich die Frage, wie dennoch konkrete Objekte und Feature Structures integriert werden können.

Betrachtet man Feature Structures wie in Abschnitt 2.3 eingeführt als Beschreibungen von Objekten oder Objektgraphen bzw. als Objektsichten (vergleiche Abschnitt 2.2.2), also Bedingungen an mögliche für sie einsetzbare Objekte, so wird der Zusammenhang zwischen konkreten Objekten und Feature Structures klarer. Geht man von einer (unendlichen) Menge möglicher Objekte aus, so kann man sich Typen sowie Feature Structures als Prädikate vorstellen, die eine Teilmenge von Objekten selektieren und die übrigen ausschließen. Ein konkretes Objekt kann also als die Feature Structure interpretiert werden, die *genau ein konkretes Objekt*

selektiert. Unifikation wird in dieser Vorstellung von einer *Vereinigung* von Information zu einer *Schnittmenge* der selektierten Objektmengen. Damit ist klar, daß eine Unifikation einer Feature Structure aus einem konkreten Objekt mit einer anderen Feature Structure entweder die konkrete Feature Structure ergibt oder scheitert.

Während für die Gleichheit primitiver (Java-)Typen klare Regeln gelten, ist zu diskutieren, wann zwei (Java-)Objekte gleich sind. Bei Objekten gibt es den Unterschied zwischen Identität und Gleichheit, der in Java durch den Vergleichsoperator `==` und die Methode `equals()` unterstützt wird, die jedes Objekt kennt. Die Method `equals()` wird in der Klasse `java.lang.Object` so vordefiniert, daß Objekte standardmäßig genau dann gleich sind, wenn sie identisch sind. Die `equals()`-Methode sollte nur in solchen Klassen überschrieben werden, in denen Kopien nicht unterscheidbar sind. Objekte solcher Klassen sollten (außer während der Konstruktion) nicht modifizierbar sein, da sonst ein Unterschied zwischen Kopien feststellbar wäre. Leider halten nicht einmal die Java-Standard-Bibliotheken diese Regel ein. Dennoch wird in der vorliegenden Implementierung davon ausgegangen, daß zwei Java-Objekte, die behaupten, gleich zu sein, dies auch über ihre gesamte Lebenszeit bleiben. Demnach wird Unifikation von Java-Objekten über die `equals()`-Methode realisiert.

Um Java-Objekte in den Feature-Structure-Graph mit aufnehmen zu können, müssen diese als Knoten (Node) gekapselt werden. Deshalb wurde auch der Knoten wie in Abbildung 2.27 gezeigt als Schnittstelle entworfen.

Abbildung 2.29 zeigt ein Entwurfs-Klassendiagramm der Typ- und Knotenklassen, das die Einbindung von Java-Typen berücksichtigt. Auf die technischen Details kann wiederum nur kurz eingegangen werden. Die Schnittstelle JavaType wird für primitive Typen (BasicType) und Java-Objekte (JavaObject) implementiert. Primitive Typen können zusätzlich Bereiche (Mengen) darstellen, um die Ausdrucksfähigkeit von Regeln zu erhöhen. Die Klasse JavaObject stellt durch Implementierung mehrerer Schnittstellen gleichzeitig einen Typ *und* einen Knoten dar. Jedes Java-Objekt wird wie oben motiviert als ein (sehr spezieller) Typ aufgefaßt. Normalerweise kommt dieser Typ aber nur an einem einzigen Knoten vor, so daß durch die direkte Implementierung der Node-Schnittstelle gekapselte Java-Objekte direkt als Knoten in Feature Structures eingesetzt werden können.

In Abbildung 2.27 sind in der Klasse FeatureStructure die Methoden canInstantiate(), instantiate() und unpackingAt() dargestellt, die bisher nicht kommentiert wurden. Diese beziehen sich auf die Integration von Java-Objekten.

unpackingAt() verhält sich genau wie at(), es sei denn, der Wurzelknoten der Substruktur ist ein JavaObject. In diesem Fall wird nicht eine Feature Structure, die das Java-Objekt beschreibt, sondern das Java-Objekt selbst geliefert.

Die Methoden canInstantiate() und instantiate() der Klasse FeatureStructure beziehen sich auf die oben angedeutete Möglichkeit, durch der Introspektionsmechanismus von Java Objekte manipulieren zu können, deren Typ erst zur Laufzeit bekannt ist. Oben wurde für die Klasse JavaConcept beschrieben, wann deren gleichnamige

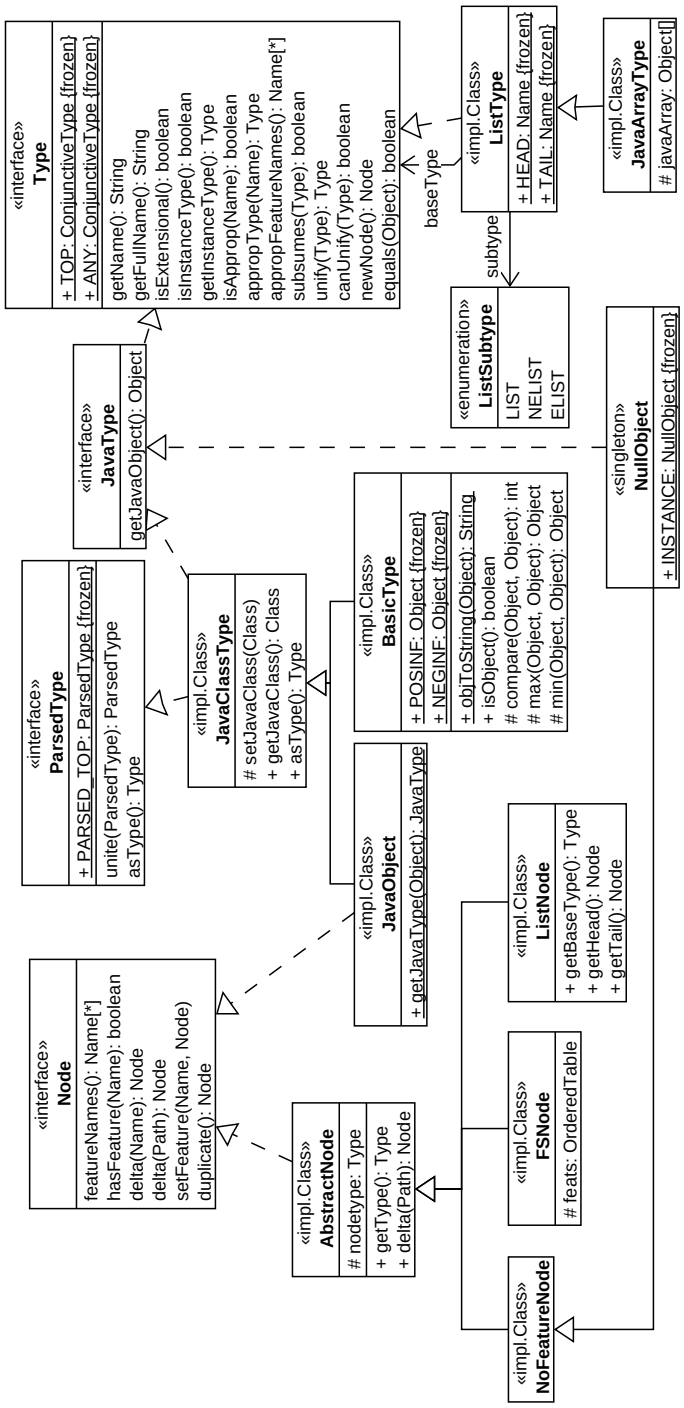

Abbildung 2.29: Entwurfs-Klassendiagramm mit Einbindung von Java-Typen.

Methode canInstantiate() *wahr* zurückliefert. Es kann jeden im Feature-Structure-Typsystem reflektierten Java-Typ in einer von zwei Arten geben, die durch die Methode isInstanceType() in Type abfragbar ist.[4] Ein Java-Typ, der ein *Exemplartyp* ist (isInstanceType() liefert *wahr*), soll in der fertig konstruierten Feature Structure instantiiert werden. Eine Feature Structure ist dann instantiierbar, wenn alle Knoten, die von einem Exemplartyp sind, instantiierbar sind. Ansonsten liefert die Methode canInstantiate() der Klasse FeatureStructure *falsch* und ein Aufruf der Methode instantiate() führt zu einer Ausnahmebedingung.

Feature Structures mit Exemplartypen dienen der komfortablen Instantiierung einzelner Java-Objekte oder im allgemeinen Fall der Erzeugung komplexer Objektgraphen. In der graphischen Darstellung werden Exemplartypen wie in UML (siehe Abschnitt 2.1.1) durch Unterstreichung von den üblichen Feature Structures unterschieden. Der in Abbildung 2.10 gezeigte Objektgraph kann als Feature Structure mit Exemplartypen angegeben werden und erzeugt bei der Instantiierung die in Abbildung 2.9 gezeigte GUI.

Damit bestimmte Objektgraphen mit dem Feature-Structure-System erzeugt werden können, müssen die gewünschten Java-Klassen zum Teil angepaßt werden. Für einige der Java-Bibliotheksklassen müssen beispielsweise entsprechende Subklassen entworfen werden, damit ein Konstruktor ohne Argumente vorhanden ist (siehe oben) und die Assoziationen und Attribute durch standardgemäß benannte Methoden gesetzt werden können.

Es ist durch die in diesem Abschnitt vorgestellten Erweiterungen möglich,

- durch *Namensräume* Typsysteme aus Teilmodellen zusammenzusetze und auf einen Namensraum mit dem Schlüsselwort `access` zuzugreifen,
- Zahlen, Zeichenketten und andere *Basistypen* in Feature Structures zu verwenden,
- beliebige Java-*Klassen und -Schnittstellen* als Konzepte zur Konstruktion von Typen zu verwenden, welche direkt in Feature Structures oder bei der Definition von Featuretypen verwendet werden können,
- beliebige Java-*Objekte* als Werte in Feature Structures zu verwenden (wenn dies zu wohltypisierten Feature Structures führt),
- Objektgraphen zu beschreiben und beispielsweise als Muster für konkrete Java-Objekte zu verwenden oder zu unifizieren und schließlich
- Java-Objektgraphen mit Exemplartypen abstrakt zu beschreiben und anschließend zu instantiieren.

[4] Zu diesem Ansatz hat Olaf Kummer maßgeblich beigetragen.

Damit werden Feature Structures für den praktischen Einsatz in Kooperation mit Java zu einem mächtigen Werkzeug. In Abschnitt 4.5 wird gezeigt, wie diese Eigenschaften in einer integrierten Modellierungsumgebung eingesetzt werden können.

2.4.3 Unifikationsalgorithmus

In Abschnitt 2.3 wurden Definitionen zu Feature Structures gegeben, die über einem hierarchischen Typsystem definiert werden. Andere Erweiterungen wie Negation ([Carpenter 1992, Zeller und Snelting 1997]) oder Disjunktion ([Strömbäck 1991, Strömbäck 1992]) werden hier nicht betrachtet.

Im vorherigen Abschnitt wurde dargestellt, daß Feature Structures hier als Objektgraphen implementiert wurden. Im wesentlichen ergibt sich der Unifikationsalgorithmus aus den Definitionen 2.35 (untypisierte Unifikation) und 2.45 (wohltypisierte Unifikation), für die auf Definition 2.43 der Funktion typify zurückgegriffen wird.

Für die untypisierte Unifikation muß die in Definitionen 2.35 gegebene Äquivalenzrelation $\bowtie$ über der Vereinigung der Knotenmengen der beiden Feature Structures etabliert werden. Im weiteren werden die mathematischen Bezeichner aus der Definition zur Beschreibung des Algorithmus und der Implementierung verwendet. Für die Implementierung wurden wie in Renew (siehe Abschnitt 3.5) die Collection-Klassen von Lea ([Lea 1998]) verwendet. Für den folgenden Pseudo-Code behandeln wir Objekte von Klassen, welche die Schnittstelle `collections.Map` implementieren, als Funktionen und analog für `collections.Set` als Mengen mit den üblichen mathematischen Relationen und Operatoren, für die es in der Collection-Bibliothek die entsprechenden Methoden gibt.

Die Relation $\bowtie$ wird in der Implementierung durch eine eigene Klasse realisiert (`de.uni_hamburg.fs.EquivRelation`), um nicht die FeatureStructure- oder FSNode-Klasse durch zusätzliche Attribute und Methoden zu vergrößern, die nur temporär (während der Unifikation) benötigt werden. Die Unifikation könnte alternativ durch statische (`static`) Methoden implementiert werden. Dies wäre aber kein sauberer Entwurf im Sinne der Objektorientierung, und es müßten viele Parameter von einer statischen Methode zur nächsten weitergereicht werden.

Abbildung 2.30 zeigt die Klasse EquivRelation in UML-Notation. Die abstrakte Klasse *ToDoItem* sowie die Klassen UnifyItem und RetypeItem sind als innere Klassen von EquivRelation implementiert.

Ein EquivRelation-Objekt existiert für die Dauer einer Unifikation und speichert die Äquivalenzrelation $\bowtie$ durch eine Abbildung (tie: `collections.UpdatebleMap`) auf neue Knoten, die eine Äquivalenzklasse repräsentieren. In der Definition werden die Äquivalenzklassen gleichzeitig als neue Knoten verwendet; dies ist jedoch für die Implementierung ungünstig, da Mengen und Knoten unterschiedliche Objektklassen sind. Für jeden neuen Knoten wird stattdessen die Menge der Knoten verwaltet, aus denen er entstanden ist (eit: `collections.UpdatableMap`). Dadurch kann sowohl abgefragt werden, zu welcher Äquivalenzklasse ein Knoten q aus $\mathrm{Q} \cup \mathrm{Q}'$ gehört

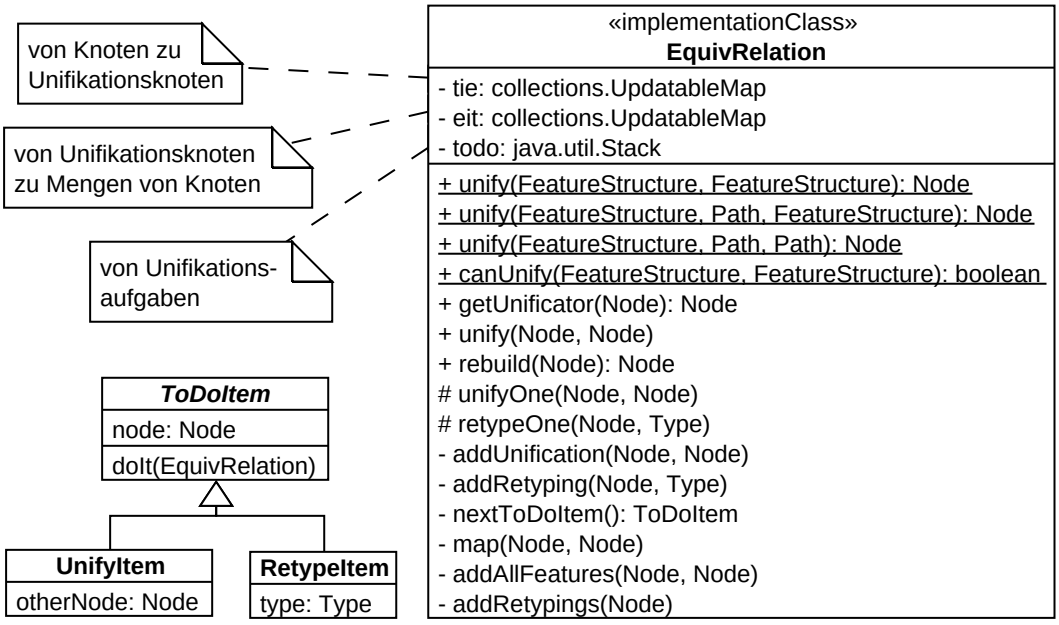

Abbildung 2.30: Die Implementierung der Unifikation als Klasse EquivRelation mit Hilfsklassen.

($\mathsf{tie}(q)$ entspricht $q^{\bowtie} = [q]_{\bowtie}$), als auch, aus welchen Knoten eine Äquivalenzklasse besteht ($\mathsf{eit}(q^{\bowtie})$ entspricht $\{q_1, q_2, \ldots, q_n\} = [q]_{\bowtie}$).

Die Unifikationsmethoden werden als statische Methoden angeboten, um dem Klienten das Instantiieren eines EquivRelation-Objekts zu ersparen. Es werden die drei Versionen der Unifikation angeboten, die im vorherigen Abschnitt für Feature Structures eingeführt wurden: Die normale Unifikation von zwei Feature Structures, die Unifikation mit einer Substruktur und die Unifikation von zwei Substrukturen derselben Feature Structure. Auch die `canUnify()`-Methode findet sich hier als statische Methode wieder. Die Implementierung ist trivial: Es wird `unify()` aufgerufen und die Ausnahme wird abgefangen. Tritt keine Ausnahme auf, wird `true` zurückgegeben, sonst `false`. Die restlichen Methoden sind Exemplarmethoden (nicht statisch) und werden während der Unifikation aufgerufen.

Nach der Konstruktion in Definition 2.35 handelt es sich beim Aufbau der Äquivalenzrelation um eine rekursive Problemstellung. Deshalb liegt es nahe, die Unifikation durch eine rekursive Methode zu implementieren. Diesem Ansatz wurde auch zunächst gefolgt, doch bei Koreferenzen und vor allem bei zyklischen Strukturen stößt man schnell auf Probleme. Durch die Rekursion ist es schwierig, einen „globalen“ Zustand zu verwalten, der den aktuellen Zustand der Äquivalenzrelation und der noch auszuführenden Knotenunifikationen korrekt widerspiegelt.

Als Alternative wird im vorliegenden Algorithmus ein Aufgaben-Stapelspeicher verwaltet, der für jede noch auszuführende Erweiterung der Äquivalenzrelation ein Aufgabenobjekt enthält. Hierfür ist die Klasse UnifyItem zuständig (die Klasse RetypeItem wird für die wohltypisierte Unifikation eingesetzt und weiter unten besprochen). Ein Objekt der Klasse UnifyItem kapselt zwei Knoten, deren Unifikation über die Methode doIt() ausgelöst werden kann. Um die Unifikation durchführen zu können, muß das aktuelle Äquivalenzrelationsobjekt übergeben werden. Das UnifyItem-Objekt delegiert den Aufruf an die Methode unifyOne() von EquivRelation. Es dient also nur der *Speicherung der Unifikationsaufgabe*.

Im folgenden Algorithmus wird ein Pseudo-Code benutzt, der die mathematischen Bezeichner verwendet. Da es aber um eine objektorientierte Implementierung geht, handelt es sich bei den Knoten um Objekte, so daß die im vorherigen Abschnitt angegebenen Methoden verwendet werden können. Dort wurden allerdings nur die normalerweise verwendeten Methoden zum *Abfragen* von Typen und Feature-Werten eingeführt. Da im Unifikationsalgorithmus Knoten konstruiert werden müssen, benötigen wir die in Abbildung 2.27 gezeigt Methode $\mathsf{setFeature}(f, q)$ in der Schnittstelle Node, die in einem Knoten ein Feature einfügt oder ändert. Diese Methode darf *nur während der Konstruktion* einer Feature Structure aufgerufen werden.

Der Algorithmus berücksichtigt nicht die im vorherigen Abschnitt diskutierten Erweiterungen um Java-Typen und -Objekte. Die in der dort geführten Diskussion gemachte Aussage, daß Java-Objekte über die `equals()`-Methode unifiziert werden, stellt eine gewisse Vereinfachung dar. Bei der Unifikation verschiedener Arten von Typen wird durch Polymorphismus die angemessene Unifikationsmethode gewählt. Details können der technischen Dokumentation der Programmquellen entnommen werden.

Mit diesen Annahmen kann der Algorithmus zur Unifikation zweier Feature Structures, die wie in Definition 2.35 gegeben sind, folgendermaßen formuliert werden.

1. Initialisiere die Abbildungen tie und eit folgendermaßen: $\forall q \in \mathrm{Q} \cup \mathrm{Q}' : \mathsf{tie}(q) := q$ und $\mathsf{eit}(q) := \{q\}$. Setze die Aufgabenliste todo auf die Liste mit dem Element $(\bar{q}, \bar{q}')$.

2. Sei (q, q') das erste Element der Aufgabenliste todo. Entferne dieses Element aus der Liste.

3. Sei $(q_{\bowtie}, q'_{\bowtie}) := (\mathsf{tie}(q), \mathsf{tie}(q'))$.

4. Ist $q_{\bowtie} = q'_{\bowtie}$, gehe zu Schritt 9.

5. Erzeuge einen neuen Knoten $\hat{q}_{\bowtie}$ vom Typ $q_{\bowtie}.\mathsf{getType}().\mathsf{unify}(q'_{\bowtie}.\mathsf{getType}()))$. Bei einer Ausnahmebedingung scheitert die Unifikation, terminiere.

6. Setze $\mathsf{eit}(\hat{q}_{\bowtie}) := \mathsf{eit}(q_{\bowtie}) \cup \mathsf{eit}(q'_{\bowtie})$ und für alle $\hat{q} \in \mathsf{eit}(\hat{q}_{\bowtie})$ setze $\mathsf{tie}(\hat{q}) := q^{\bowtie}$.
7. Für alle Features f, für die es ein beliebiges $\hat{q} \in \mathsf{eit}(\hat{q}_{\bowtie})$ gibt, so daß $\hat{q}.\mathsf{hasFeature}(f)$ gilt, setze $\hat{q}_{\bowtie}.\mathsf{setFeature}(f, \hat{q}.\mathsf{delta}(f))$.
8. Für alle Features f und alle $\hat{q}, \hat{q}' \in \mathsf{eit}(\hat{q}_{\bowtie})$, für die gilt $\hat{q} \neq \hat{q}'$ und $\hat{q}.\mathsf{hasFeature}(f)$ sowie $\hat{q}'.\mathsf{hasFeature}(f)$, füge den Knoten $(\hat{q}.\mathsf{delta}(f), \hat{q}'.\mathsf{delta}(f))$ vorne in die Aufgabenliste todo ein.
9. Wenn die Aufgabenliste todo nicht leer ist, gehe zu Schritt 2.
10. Wenn die Aufgabenliste todo leer ist, terminiere.

Durch die Organisation der Aufgabenliste als Stapelspeicher ist sichergestellt, daß zwei Knoten nicht mehrfach identifiziert werden. Wenn Knoten in der Aufgabenliste auftreten, die bereits identifiziert sind, wird in Schritt 4 der Schleifenrumpf übersprungen.

Nachdem die Äquivalenzrelation durch den Algorithmus gefunden ist, muß wie in Definition 2.35 die Ergebnis-Feature-Structure konstruiert werden. Dazu werden ab der neuen Wurzel $\mathsf{tie}(\bar{q})$ alle Kanten nach folgendem Schema ersetzt:

$$\hat{q}.\mathsf{setFeature}(f, \mathsf{tie}(\hat{q}.\mathsf{delta}(f)))$$

Da zyklische Feature Structures erlaubt sind, muß im Rekonstruktionsalgorithmus explizit dafür gesorgt werden, daß dieses Schema für jeden erreichbaren Knoten für alle Features genau einmal angewandt wird. Dies ist leicht zu realisieren über eine Menge bisher bearbeiteter Knoten, die bei jeder Anwendung aktualisiert wird.

Als alternativer Unifikationsalgorithmus könnten auch erst bei der Rekonstruktion die Knotentypen unifiziert werden. Der Algorithmus wurde aber bewußt wie angegeben gewählt, da so ein Scheitern der Unifikation eher erkannt wird. Dies ist erwünscht, da vor allem im Petrinetz-Simulator (siehe Abschnitt 4.5) viele Unifikationen „ausprobiert“ werden, die oft scheitern.

Da der in dieser Arbeit vorgestellte Feature-Structure-Formalismus eine wohltypisierte Variante der Feature Structures definiert, wird diese auch in der Implementierung unterstützt. Man könnte hierfür die in Definition 2.43 konstruktiv definierte Funktion typify implementieren und auf das Ergebnis der Unifikation anwenden. In Abschnitt 2.3.3 wurde gezeigt, daß diese Operation scheitern kann, wenn typify nicht definiert ist. Deshalb wurde der oben angegebene Unifikationsalgorithmus so erweitert, daß auch hier möglichst früh ein Scheitern erkannt wird.

Die Aufgabenliste besteht im Unifikationsalgorithmus oben nur aus einer Art von *ToDoItem*s, nämlich aus UnifyItems. Die in Abbildung 2.30 gezeigte weitere Subklasse RetypeItem dient der Anwendung von Feature-Typisierungs-Regeln während der Unifikation. Immer, wenn in Schritt 5 durch den Join ein neuer Typ entsteht, werden die von diesem Typ forcierten Feature-Typisierungen auf alle in Schritt 7 errechneten Folgeknoten angewandt.

Die Aufgabenliste besteht damit aus zwei verschiedenen Arten von *ToDoItem*s. Durch die gemeinsame abstrakte Oberklasse kann Polymorphismus genutzt werden, so daß immer nur die doIt()-Methode aufgerufen werden muß und eine explizite Fallunterscheidung oder eine Unterteilung in zwei Aufgabenlisten entfällt.

Kapitel 3

Prozeßorientierte Modellierung

In Kapitel 2 wurden verschiedene Konzepte und Techniken der informationsorientierten Modellierung vorgestellt. Dabei wurde bewußt betont, daß es sich bei den Modellierungszielen um Daten- oder Informationsobjekte handelt.

Die Objektorientierung (siehe u.a. [Coad und Yourdon 1991, Rumbaugh et al. 1991, Meyer 1997]) hat die Kombination aus Daten und Funktionen und damit die Kapselung von Daten durch Funktionen der abstrakten Datentypen ([Astesiano et al. 1995]) zum Vorbild. Hier spricht man statt von Funktionen von Operationen, was den prozeduralen Charakter betont. Operationen auf Objekten gehen insofern über Funktionen hinaus, als sie Seiteneffekte auslösen können, indem sie den Zustand des Objekts ändern. Im Gegensatz dazu entstehen bei Funktionsaufrufen neue Datenobjekte. Aus der Anwendung von Operationen entstehen Prozesse, in deren Verlauf sich der Zustand bzw. die Daten der beteiligten Objekte ändern.

Feature Structures wurden in Abschnitt 2.3.1 als Repräsentanten von Information oder Daten eingeführt. Im folgenden wird dafür argumentiert, daß der Informationsbegriff so allgemein ist, daß auch Operationen und sogar Prozesse als Informationseinheiten modelliert werden können. Dazu wenden wir uns zunächst der deklarativen Programmierung zu. Aus der Künstlichen Intelligenz (KI) ist die Idee des *general problem solvers* (GPS) bekannt. Hier wird die Umgebung („Welt“) durch Zustände beschrieben, während Operationen durch Regeln dargestellt werden, welche die Zustandsänderungen beschreiben. Damit können auch Regeln auf einer Meta-Ebene wieder als Informationsobjekte aufgefaßt werden.

In der Objektorientierung sind durch die Unified Modeling Language (UML) die Zustandsdiagramme von Harel (siehe [Harel 1987]) zu einem Standard geworden. Sie werden üblicherweise eingesetzt, um den Lebenszyklus eines einzelnen Objekts zu beschreiben. Die Ereignisse (*events*), die ein Objekt empfängt, führen zu einem Zustandsübergang. Zustandsdiagramme lassen sich natürlich auch durch eine Datenstruktur beschreiben. Eine operationale Semantik kann man ihnen, genau wie den deklarativen Programmen, zuordnen, indem man die möglichen Prozesse des

zugehörigen Zustandsmodells betrachtet.

Zustandsdiagramme sind in UML und bereits von Harel in einer Form definiert worden, die über einen endlichen Automaten hinausgeht. Da durch diese Erweiterungen Nebenläufigkeit möglich wird, bieten sich Petrinetze zu ihrer formalen Beschreibung an ([Moldt 1996]). Durch ihre operationale Semantik vermögen Petrinetze nicht nur die Struktur von Zustandsdiagrammen, sondern auch deren Ausführung formal zu beschreiben.

In den folgenden Abschnitten wird auf verschiedene prozeßorientierte Modellierungskonzepte und -techniken näher eingegangen, insbesondere aber auf Petrinetze. Während im vorherigen Kapitel formale Definitionen zu Typsystemen und Feature Structures direkt im Text angegeben sind, finden sich Definitionen zu Petrinetzen in Anhang B. Dieser Aufbau wurde gewählt, da davon auszugehen ist, daß Petrinetze in der Informatik bekannter sind als Feature Structures. Weiterhin werden im Bereich der Feature Structures einige modifizierte Definitionen und Notationen benutzt, die ausführlicher eingeführt und deren Eigenschaften gezeigt werden müssen. Im Bereich der prozeßorientierten Modellierung werden erst im nächsten Kapitel eigene Definitionen aufgestellt, die FSNets formal beschreiben. In diesem Kapitel greifen wir auf die bestehenden Petrinetz-Definitionen zurück.

Um praxisbezogen zu beginnen, werden zunächst in Abschnitt 3.1 verschiedene Konzepte und Notation verschiedener prozeßorientierter Modellierungstechniken vorgestellt. Dazu gehören die dynamischen Modellierungstechniken von UML und verschiedene Techniken aus dem Bereich der Modellierung von Geschäftsprozessen, der einen Schwerpunkt des Einsatzes von Prozeßmodellierungstechniken in der Praxis darstellt. In den folgenden Abschnitten widmen wir uns den Petrinetzen, wobei verschiedene Varianten vorgestellt werden. Neben den klassischen B/E- und S/T-Netzen gehören die gefärbten Petrinetze zum Standardrepertoire, das in Abschnitt 3.2 behandelt wird. Einen besonderen Einfluß auf den in dieser Arbeit vorgestellten Ansatz haben verschiedene Varianten von Netzen in Netzen (Abschnitt 3.3), die aktive Marken in Petrinetzen erlauben. Hier werden elementare Objektsysteme, Linearlogische Petrinetze und Referenznetze vorgestellt. Motiviert wird diese Erweiterung der klassischen Petrinetze durch Einflüsse aus der Objektorientierung. Sodann wird in Abschnitt 3.4 die Geschäftsprozeßmodellierung als ein Anwendungsgebiet für Petrinetz-Modellierung betrachtet. Der letzte Unterabschnitt 3.5 dieses Abschnitts stellt ein von Kummer und dem Autoren entwickeltes Software-Werkzeug vor, mit dem Referenznetze editiert und simuliert werden können. Renew ist das bisher einzige Petrinetz-Werkzeug, in dem das Netze-in-Netzen-Paradigma praktisch umgesetzt wurde.

3.1 Konzepte und Notationen der prozeßorientierten Modellierung

Die Konzepte und Notationen für prozeßorientierte Modellierungstechniken stammen vor allem aus zwei Bereichen. Zum einen widmen sich Software-Modellierungstechniken wie UML, StateCharts und viele andere mehr und mehr der Modellierung der Systemdynamik, zum anderen werden Geschäftsprozesse modelliert, simuliert und automatisiert.

Beide Bereich haben eine große Anzahl von Modellierungstechniken hervorgebracht, von denen in den folgenden Abschnitten einige ausgewählte Vertreter vorgestellt werden.

Die dynamischen und prozeßorientierten Techniken von UML werden in den ersten drei Abschnitten vorgestellt. Abschnitt 3.1.1 gibt eine kurze Einordnung und grenzt die hier genauer betrachteten Prozeßmodellierungstechniken von Interaktionsdiagrammen wie Sequenz- und Kollaborationsdiagrammen ab. Abschnitt 3.1.2 stellt die Zustandsdiagramme und Abschnitt 3.1.3 die Aktivitätsdiagramme von UML ausführlicher vor.

In Abschnitt 3.1.4 werden die Aktivitätenmodelle von FlowMark und in Abschnitt 3.1.5 ereignisgesteuerte Prozeßketten (EPK) nach Scheer als repräsentative Techniken vorgestellt.

Petrinetze stellen einen der Schwerpunkte dieser Arbeit dar und werden in mehreren folgenden Abschnitten ab 3.2 ausführlich diskutiert, wobei auf besondere Eigenschaften in Hinblick auf Geschäftsprozeßmodellierung eingegangen wird.

3.1.1 Modellierung von Dynamik in UML

UML erlaubt neben der in Abschnitt 2.1.1 vorgestellten statischen Modellierung auch die Modellierung der *Dynamik* eines objektorientierten Systems. Dies entspricht einer weiteren der in Abschnitt 1.2 beschriebenen Sichten.

In UML werden verschiedene Arten der Modellierung von Dynamik unterschieden ([OMG 2000]). Die in Abschnitt 2.2.2 bereits erwähnten Kollaborationen bilden die Grundlage für *Sequenz-* und *Kollaborationsdiagramme*, die mit dem Oberbegriff *Interaktionsdiagramm* zusammengefaßt werden. Beide Diagrammarten beschreiben einen mehr oder weniger konkreten *Ablauf* des zu spezifizierenden Systems, der in UML als *Interaktion* bezeichnet wird[1]. Interaktionen beschreiben den Nachrichtenaustausch auf der Basis einer Kollaboration, also zwischen einer Menge von Objekten bzw. Objektsichten (siehe Abschnitt 2.2.2), die in Assoziationsbeziehungen stehen.

[1] In Abschnitt 1.1.2 wird der Begriff *Interaktion* für die Kommunikation zwischen zwei Objekten verwendet. Eine Interaktion nach UML ist im Vergleich dazu eine Verallgemeinerung in bezug auf die Anzahl der beteiligten Objekte. In [Engels et al. 2000] wird aber argumentiert, daß UML in bezug auf die dargestellten Interaktionsmuster weniger ausdrucksstark ist.

In Sequenzdiagrammen liegt der Fokus auf der Beschreibung der *Reihenfolge* der Kommunikationsschritte, wogegen Kollaborationsdiagramme den Nachrichtenaustausch zwischen Objekten und damit die Objektschnittstellen betonen.

Da Interaktionen Abläufe des Systems durch Beispiele darstellen, nicht durch Modelle, die solche Abläufe hervorbringen können, handelt es sich um eine Form des Modellierens durch Beispiele. Das Ziel dieser Arbeit ist die Definition einer *ausführbaren* Modellierungstechnik, so daß sich Interaktionen des Systems aus der Simulation bzw. Ausführung des Modells ergeben. Interaktionsdiagramme könnten demnach analog zu Schaltfolgen oder Prozessen von Petrinetzen (siehe Abschnitt 3.2.2) aus einem gegebenen ausführbaren Modell generiert werden. Diese Möglichkeit geht aber über das Ziel dieser Arbeit hinaus, weshalb im folgenden Interaktionen und Interaktionsdiagramme nicht näher betrachtet werden.

Die Modelle bzw. Diagramme in UML, welche das dynamische Verhalten des Systems beschreiben, ohne dafür auf Beispielabläufe zurückzugreifen, sind die in den folgenden Abschnitten vorgestellten Zustands- und Aktivitätsdiagramme.

3.1.2 Zustandsdiagramme in UML

Ein Zustandsdiagramm modelliert die Dynamik des Systems durch Zustände und Ereignisse, die einzelnen Objekten zugeordnet werden. Ein Zustandsdiagramm zeigt demnach *Intra*objektverhalten, während zur Spezifikation von *Inter*objektverhalten Kollaborationen oder die unten vorgestellten Aktivitätsdiagramme verwendet werden.

Zustandsdiagramme in UML haben ihren Ursprung in den *StateCharts* von Harel ([Harel 1987]). StateCharts und damit auch UML-Zustandsdiagramme stellen eine Verallgemeinerung der Konzepte von endlichen Automaten nach Moore und Mealy (siehe z.B. [Conway 1971]) dar. Auch die in Abschnitt 3.2 vorgestellten Petrinetze können als Verallgemeinerung endlicher Automaten angesehen werden. Im folgenden werden Zustandsdiagramme in Anlehnung an [Hitz und Kappel 1999] eingeführt.

Zustandsdiagramme werden in der objektorientierten Modellierung sowohl zur Darstellung des *Objekt-Lebenslaufs* (*object life-cycle*) als auch zur genaueren Spezifikation einzelner Methoden genutzt. Je nach Granularitätsebene der Objektmodellierung kann ein Zustandsdiagramm damit die Dynamik eines gesamten Systems oder Teilsystems beschreiben. Im Sinne dieser Arbeit handelt es sich also um eine prozeßorientierte Modellierungstechnik.

Ein Zustandsdiagramm wird als ein Graph dargestellt, wobei die Knoten *Zustände* (*states*) und die Kanten *Zustandsübergänge*[2] (*state transitions*) repräsentieren. Wie in endlichen Automaten werden Zustandsübergänge durch Eingaben ausgelöst, die in Zustandsdiagrammen als *Ereignisse* (*events*) bezeichnet werden.

[2]Der in [Hitz und Kappel 1999] als Alternative angegebene Begriff *Transition* wird in dieser Arbeit nicht verwendet, um Verwechslungen mit Transitionen in Petrinetzen zu vermeiden.

Zustandsübergänge können während ihres Auftretens auszuführende *Aktionen* zugeordnet bekommen. Während das System konzeptionell eine gewisse Zeit in einem Zustand verweilen kann, werden Zustandsübergänge als zeitlos und unteilbar betrachtet. Der Zustandsübergang und die mit ihm verbundenen Aktionen verbrauchen also konzeptionell keine Zeit und können nicht scheitern, was z.B. durch Fehler bei der Ausführung einer Aktion vorstellbar wäre. Einem Zustand kann eine *Aktivität* zugeordnet werden, welche solange ausgeführt wird, wie sich das System in diesem Zustand befindet. Aktivitäten haben also im Gegensatz zu Aktionen eine zeitliche Ausdehnung und können unterbrochen werden.

Ein Zustand wird in UML als abgerundeter Kasten notiert, der durch eine horizontale Linie unterteilt wird. Über der Linie wird der (optionale) Name des Zustands angegeben, während unter der Linie die Aktivität angegeben und sogenannte *innere Transitionen* angegeben werden, die einen Sonderfall von Zustandübergangs-Schleifen darstellen. Werden für einen Zustand keine Aktivitäten und inneren Transitionen angegeben, kann die horizontale Linie entfallen.

Zustandsübergänge werden durch gerichtete Kanten von einem Quell- zu einem Ziel-Zustand dargestellt und können durch folgende Angaben beschriftet werden:

- ein *auslösendes Ereignis*, das formale Parameter als Argumente in runden Klammern spezifizieren kann,
- eine optionale *Überwachungsbedingung*, die beim Eintreffen des Ereignisses erfüllt sein muß, damit der Zustandsübergang stattfinden kann, und in eckigen Klammern notiert wird,
- *Aktionen*, die im Zuge des Zustandsübergangs auszuführen sind und hinter einem Schrägstrich folgen und
- eine optionale Transitionszeit, deren Syntax und Semantik in [OMG 2000] nicht ausreichend genau spezifiziert wird.

Bei den Aktionen handelt es sich um eine Sequenz von Befehlen in Pseudo-Code, der in der UML-Spezifikation nicht weiter eingeschränkt wird. Eine Sonderstellung nimmt jedoch das Versenden von Ereignissen an die Zustandsmodelle anderer Objekte ein, für das eine eigene, dem objektorientierten Methodenaufruf nachempfundene Syntax vorgesehen ist (siehe [OMG 2000]).

Die Ablaufsemantik von UML-Zustandsdiagrammen wird in [OMG 2000] nicht formal definiert, läßt sich aber wie folgt zusammenfassen. Wenn ein Ereignis eintrifft, werden die Zustandsübergänge als Kandidaten bestimmt, welche dieses Ereignis als Auslöser tragen *und* in deren Quell-Zustand sich das System befindet. Von allen Kandidaten, deren Überwachungsbedingung zu „wahr“ ausgewertet wird, wird eine nichtdeterministisch ausgewählt. Ist die Menge der Kandidaten leer, so wird das Ereignis verworfen. Eine solche Semantik ist nur mit einer sequenziellen Betrachtung eintreffender Ereignisse realisierbar.

Neben den explizit durch andere Zustandsmodelle oder die Umgebung gesendeten Ereignissen gibt es in Zustandsdiagrammen sogenannte *Pseudo-Ereignisse*. Das Pseudo-Ereignis do sorgt für den Start der Aktivität eines Zustands. Die Pseudo-Ereignisse entry und exit werden beim Betreten bzw. Verlassen eines Zustands ausgelöst. Alle bisher genannten Pseudo-Ereignisse werden sinnvollerweise nur von internen Transitionen genutzt. Ein weiteres wichtiges Pseudo-Ereignis, das Beendigungsereignis (*completion event*), wird ausgelöst, sobald eine Aktivität beendet ist. Da ein Beendigungsereignis die Standardanschrift eines Zustandübergangs darstellt, werden unbeschriftete Kanten in Zustandsdiagrammen ausgelöst, nachdem die Aktivität des Quell-Zustands beendet wurde.

Auch bei Zuständen gibt es bestimmte zusätzliche Konstrukte, die als *Pseudo-Zustände* bezeichnet werden. Ein Pseudo-Zustand besitzt – wie ein Zustandsübergang – keine zeitliche Ausdehnung, so daß sich ein System konzeptuell nie in einem Pseudo-Zustand befindet. Entsprechend können einem Pseudo-Zustand keine Aktivitäten zugeordnet werden.

Der *Startzustand* wird durch einen ausgefüllten Kreis dargestellt und mit einer Zustandsübergangskante mit dem initial zu aktivierenden Zustand verbunden.

Entscheidungsknoten werden als Rauten dargestellt und sind „syntaktischer Zucker“, um mehrere Ausgehende Zustandsübergänge mit sich gegenseitig ausschließenden Überwachungsbedingungen zu bündeln. An Ausgangskanten von Entscheidungsknoten kann das Schlüsselwort else als Überwachungsbedingung eingesetzt werden. Es entspricht einer Negation der Konjunktion der Überwachungsbedingungen aller anderen von diesem Entscheidungsknoten ausgehenden Zustandsübergänge.

Synchronisationsbalken werden meist in Aktivitätsdiagrammen verwendet und deshalb im folgenden Abschnitt vorgestellt. Zu den Pseudo-Zuständen zählen weiterhin und Synch-Zustände, Verbindungsknoten und History-Zustände, auf die hier nicht weiter eingegangen werden soll.

Der *Endzustand*, dargestellt als ausgefüllter Kreis mit einem umgebenden Ring, ist *kein* Pseudo-Zustand, da das System in ihm verweilen kann. Der Endzustand ist vielmehr ein speziell ausgezeichneter Zustand, der keine internen Transitionen oder Aktivitäten erlaubt. Wenn sich ein Endzustand innerhalb eines verfeinerten Zustands (siehe unten) befindet, löst er als implizite *entry*-Aktion ein Beendigungsereignis aus.

Um größere Modelle strukturieren zu können, gibt es in UML-Zustandsdiagrammen zwei Arten der Verfeinerung: ODER- und UND-Verfeinerung.

Die *ODER-Verfeinerung* erlaubt die Verfeinerung eines Zustands durch ein vollständiges Zustandsdiagramm. Das verfeinerte Zustandsdiagramm kann seine Verfeinerung entweder direkt enthalten (*inline expansion*) oder textuell darauf verweisen.

Zustandsübergänge, welche einen ODER-verfeinerten Zustand als Ziel haben, gelten als mit dem Zustand der Verfeinerung verbunden, der durch den Startzu-

stand festgelegt wird. Tritt ein ODER-verfeinerter Zustand als *Quelle* auf, so kann der Zustandsübergang *jeden* der Zustände der ODER-Verfeinerung unterbrechen, wenn sein auslösendes Ereignis eintrifft. Ein solches Verhalten kann in einem flachen Zustandsdiagramm durch entsprechend viele Zustandsübergänge simuliert werden. Insbesondere hier zeigt sich, daß durch ODER-Verfeinerung sehr viel kompaktere Modelle erreicht werden können.

Als Sonderfall kann man betrachten, daß die ausgehende Kante keine Anschrift trägt und somit wie oben dargestellt auf das Beendigungsereignis des (verfeinerten) Quell-Zustands wartet. Da nur der Übergang in den Endzustand der ODER-Verfeinerung ein solches Ereignis erzeugt, kann man sich die Kante auch als mit dem Endzustand verbunden vorstellen.

Weiterhin sind Zustandsübergänge erlaubt, die Zustände aus beliebigen Verfeinerungsebenen direkt verbinden. Obwohl diese Art der Modellierung den Prinzipien der Kapselung und Modularisierung widerspricht, kann sie in einigen Fällen zu kompakteren Modellen führen. Es wird hier empfohlen, solche Kanten nur sehr sparsam einzusetzen, beispielsweise, um die verschiedenen Endzustände *Erfolg* und *Fehler* eines verfeinerten Zustands auf der oberen Ebene abfragen zu können.

Eine ODER-Verfeinerung läßt sich durch syntaktische Umformungen immer auf ein flaches Zustandsdiagramm abbilden, stellt also kein ausdrucksstärkeres Modellierungsmittel dar.

Eine *UND-Verfeinerung* untergliedert die ODER-Verfeinerung ein Modell in einzelne Zustandsdiagramme, nur daß diese als *gleichzeitig aktiv* angesehen werden. Während in einem Zustandsdiagramm ohne UND-Verfeinerung immer *nur ein atomarer* (also nicht verfeinerter) Zustand zur Zeit aktiviert ist, gibt es genau einen aktiven Zustand *in jedem Unterdiagramm* der UND-Verfeinerung.

Durch die Darstellung von Nebenläufigkeit gewinnt die Zustandsmodellierung an Ausdrucksstärke, die nur durch die Verwendung der im nächsten Abschnitt beschriebenen Synchronisationsbalken adäquat modelliert werden kann. In [Hitz und Kappel 1999], S. 144, wird gezeigt, daß sich auch UND-verfeinerte Zustandsmodelle in flache Modelle abbilden lassen, daß dabei aber einige semantische Feinheiten verlorengehen. Daß eine solche Abbildung dennoch keinen allzu großen Verlust an Nebenläufigkeit bedeutet, liegt daran, daß in der Semantik von Zustandsmodellen in UML nur nebenläufige Aktivitäten, nicht nebenläufige Ereignisse betrachtet werden (siehe dazu die Diskussion von *verzögerten Ereignissen* (*deferred events*) in [Hitz und Kappel 1999]).

Wenn ein Unterzustand aktiv ist, so gilt auch der verfeinerte Zustands als aktiv. Auch ohne UND-Verfeinerung können also mehrere Zustände aktiv sein, wenn sie in einer Verfeinerungsbeziehung stehen. Als *nebenläufig aktiviert* gelten dagegen nur solche Zustände, die gleichzeitig aktiv sind, aber nicht in einer Verfeinerungsbeziehung stehen. Verfeinerungen können beliebig tief verschachtelt werden.

Abbildung 3.1 zeigt als Beispiel für eine Zustandsmodellierung eine einfache graphischen Oberfläche für eine Aufgabenliste (*to-do list*) und das zugehörige Zustands-

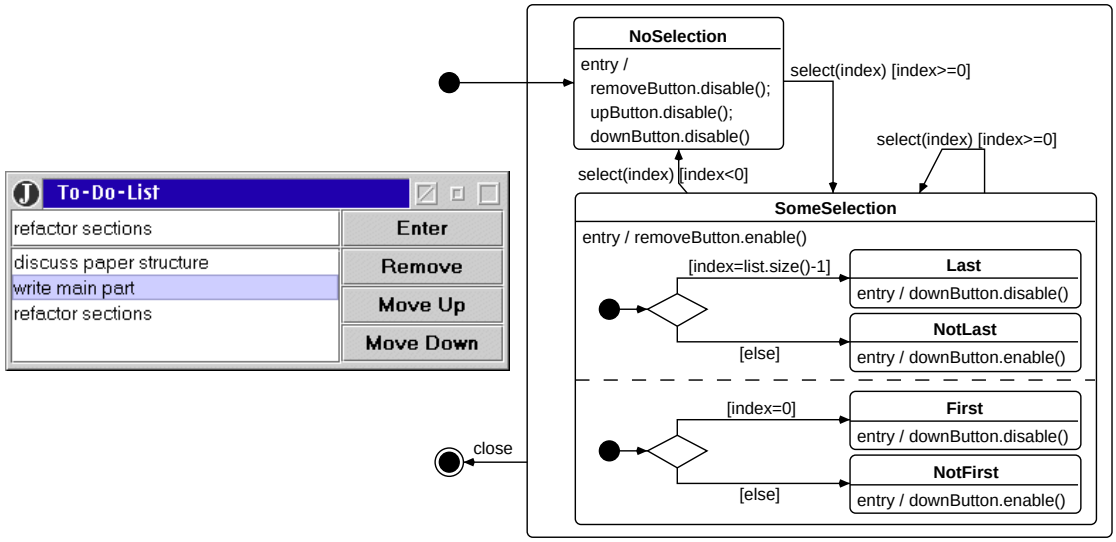

Abbildung 3.1: Eine einfache graphischen Oberfläche für eine Aufgabenliste (*to-do list*) und das zugehörige Zustandsdiagramm.

diagramm. Durch das Zustandsdiagramm wird modelliert, wann welche der Schaltflächen Remove, Move Up und Move Down verfügbar (*enabled*) sind. Das Hinzufügen neuer Einträge, das durch das Eingabefeld links oben und die Schaltfläche Enter erfolgt, wird nicht betrachtet. Es wird angenommen, daß die Selektion eines Listenelements im linken Bereich das Ereignis select(index) auslöst, wobei index der Index der selektierten Zeile von 0 bis list.size()-1 ist. Falls die Selektion zurückgenommen wird, werde das Ereignis select(-1) gesendet. Durch enable() und disable() wird der Aktivierungszustand der Schaltflächen beeinflußt, die das *to-do-list*-Objekt als removeButten, upButton und downButton kennt. Entsprechende Assoziationen könnten über ein Klassen- oder Objektdiagramm angegeben werden (siehe Abschnitt 2.1.1).

Im Beispiel aus Abbildung 3.1 kommen Start- und Endzustand, Zustände, Zustandsübergänge mit und ohne Überwachungsbedingungen, Entscheidungsknoten, *entry*-Aktionen und sowohl UND- als auch ODER-verfeinerte Zustände vor.

Die Zustände NoSelection (nichts ist selektiert) und SomeSelection (eines der Listenelemente ist selektiert) sind zu einer unbenannten ODER-Verfeinerung zusammengefaßt. SomeSelection wird durch eine UND-Verfeinerung (die weitere ODER-Verfeinerungen enthält) genauer spezifiziert.

Die UND-Verfeinerung wird hier nicht zur Modellierung von Nebenläufigkeit, sondern zur Darstellung *orthogonaler Zustände* verwendet, was eine typische Anwendung darstellt ([Hitz und Kappel 1999]). Man beachte, daß in einem aktiven UND-verfeinerten Zustand (im Beispiel SomeSelection) in jedem Unterbereich jeweils ein Subzustand aktiv ist (Last oder NotLast *und* First oder NotFirst), in einer ODER-

Verfeinerung hingegen nur genau einer. Ein Ereignis, das eine UND-Verfeinerung verläßt (im Beispiel select), muß demnach den jeweils aktiven Unterzustand jedes Teilbereichs verlassen. Bei einer ODER-Verfeinerung muß ein verlassendes Ereignis (im Beispiel close) den einzigen aktivierten Unterzustand verlassen.

An dem Beispiel wird deutlich, daß eine geschickte Zusammenfassung in verfeinerte Zustände viele Zustandsübergangskanten sparen kann.

Die Kante vom Startzustand zu NoSelection stellt ein Beispiel für einen Zustandsübergang über verschiedene Verfeinerungsebenen dar.

Wie man weiterhin sieht, können Entscheidungen mit (wie in den UND-Verfeinerungen von SomeSelection) oder ohne Entscheidungsknoten (wie beim Ereignis select(index) im Zustand SomeSelection) modelliert werden.

Die im folgenden Abschnitt behandelten Aktivitätsdiagramme stellen syntaktisch eine Variante von Zustandsdiagrammen dar, werden aber zu anderen Zwecken eingesetzt.

3.1.3 Aktivitätsdiagramme in UML

Für die Modellierung von objektübergreifenden Prozessen oder von Abläufen innerhalb einer Methode wird in UML das *Aktivitätsdiagramm* verwendet. Während das Zustandsdiagramm Verhalten zustands- und ereignisbasiert beschreibt, stellen Aktivitätsdiagramme *prozedurales* Verhalten dar. Die folgende Einführung in Aktivitätsdiagramme basiert auf [Hitz und Kappel 1999].

Die Beschreibung von prozeduralem Verhalten wird in der Analyse benötigt, um Abläufe zur spezifizieren, die keinem einzelnen Objekt zugeordnet werden können, da noch keine (verantwortlichen) Objekte festgelegt sind oder der Ablauf objektübergreifend betrachtet werden soll. Dies ist bei Analysemodellen von Geschäftsprozessen (vergleiche Abschnitt 3.4) der Fall, in der *Geschäftsobjekte* (*business objects*) betrachtet werden, nicht jedoch der Geschäftsprozeß selbst als Objekt oder Methode eines Objekts.

Im Entwurf kann mit Aktivitätsdiagrammen das prozedurale Verhalten einzelner Methoden detailliert modelliert werden. Hier ähneln Aktivitätsdiagramme den bereits in der strukturierten Analyse und später in OMT verwendeten *Datenflußdiagrammen* ([Rumbaugh et al. 1991]).

Ein Aktivitätsdiagramm in UML beschreibt einen Ablauf unter drei Sichten. Für die einzelnen Schritte kann angegeben werden, *was* diese tun, in *welcher Reihenfolge* sie auftreten und optional, welche Objekte an ihrer Durchführung beteiligt sind. Werden die „Objekte“ der dritten Sicht konkreter als Personen und Ressourcen betrachtet, ist diese Sicht vor allem für die Geschäftsprozeßmodellierung interessant, auf die wir später zurückkommen.

Ein einzelner Schritt wird in einem Aktivitätsdiagramm durch eine spezielle Variante eines Zustands modelliert, in dem eine Aktivität ausgeführt wird. Diese *Aktivitätszustände* werden in der hier verwendeten Sprechweise auch mit den Aktivitäten,

die sie ausführen, gleichgesetzt. Ein Aktivitätszustand wird (ähnlich einem Zustand) als abgerundeter Kasten dargestellt, nur daß sich die Rundungen links und rechts zu Halbellipsen treffen.

Die Aktivitäten selbst werden in UML meist nur natürlichsprachlich oder durch Pseudo-Code beschrieben. Eine formale Sprache für Aktivitäten wird von UML also nicht vorgegeben, steht aber mit der *action specification language* (ASL, [Consortium 2000]) kurz vor der Standardisierung.

Bevor die Beteiligung von Objekten an Aktivitäten vorgestellt wird, betrachten wir zunächst nur den Steuerfluß.

Steuerfluß

Um den Steuerfluß eines Ablaufs anzugeben, werden Start- und Endzustand und gerichtete Kanten ähnlich wie beim Zustandsdiagramm verwendet, die hier *Steuerflußkanten* heißen. Ereignisse sind an Steuerflußkanten nach UML syntaktisch möglich, werden jedoch in der praktischen Modellierung selten verwendet.

Der Unterschied eines Aktivitätszustands und eines (normalen) Zustands liegt darin, daß Aktivitätszustände *nicht unterbrechbar* sind. Dadurch bekommen Ereignisse an Steuerflußkanten eine etwas andere Semantik: der Übergang zum nächsten Aktivitätszustand wird vom Ereignis nicht *forciert*, sondern die Fortführung des Ablaufs wird nach Beendigung der ersten Aktivität bis zum Eintreffen des Ereignisses *blockiert*. Um diese veränderte Semantik deutlicher hervorzuheben, gibt es für Ereignisse in Aktivitätsdiagrammen eine alternative Notation (siehe dazu [OMG 2000] oder [Hitz und Kappel 1999]).

Steuerflußkanten können wie Zustandsübergänge mit Überwachungsbedingungen beschriftet sein, wobei durch die oben beschriebene Semantik diese dann ausgewertet werden, wenn die Vorgängeraktivität beendet ist. Damit der Steuerfluß weder verklemmt noch nichtdeterministisch ist, sollten sich die Überwachungsbedingungen aller Ausgangskanten eines Aktivitätszustands wechselseitig ausschließen und gemeinsam die insgesamt möglichen Fälle abdecken.

Mit den bisher eingeführten Mitteln lassen sich einfache sequenzielle Abläufe mit Verzweigungen (Entscheidungen) bereits modellieren. Zur Darstellung von Nebenläufigkeit wird ein weiterer der oben bereits erwähnten Pseudo-Zustände eingeführt, der die Notation *komplexer Steuerflußkanten* erlaubt. Komplexe Steuerflußkanten werden auch *Synchronisationsbalken* genannt, obwohl, wie in [Hitz und Kappel 1999] bemerkt wird, diese nur dann den Steuerfluß synchronisieren, wenn sie über *mehrere Eingangskanten* verfügen. Ein „Synchronisationsbalken“ mit *mehreren Ausgangskanten* spaltet im Gegenteil den Steuerfluß auf, so daß die Folgeaktivitäten nebenläufig ausgeführt werden.

Durch die ähnliche Definition von Aktivitäts- und Zustandsdiagrammen im UML-Meta-Modell sind Synchronisationsbalken (in Kombination mit UND-Verfeinerungen) auch in Zustandsdiagrammen zulässig, werden dort aber eher selten

eingesetzt.

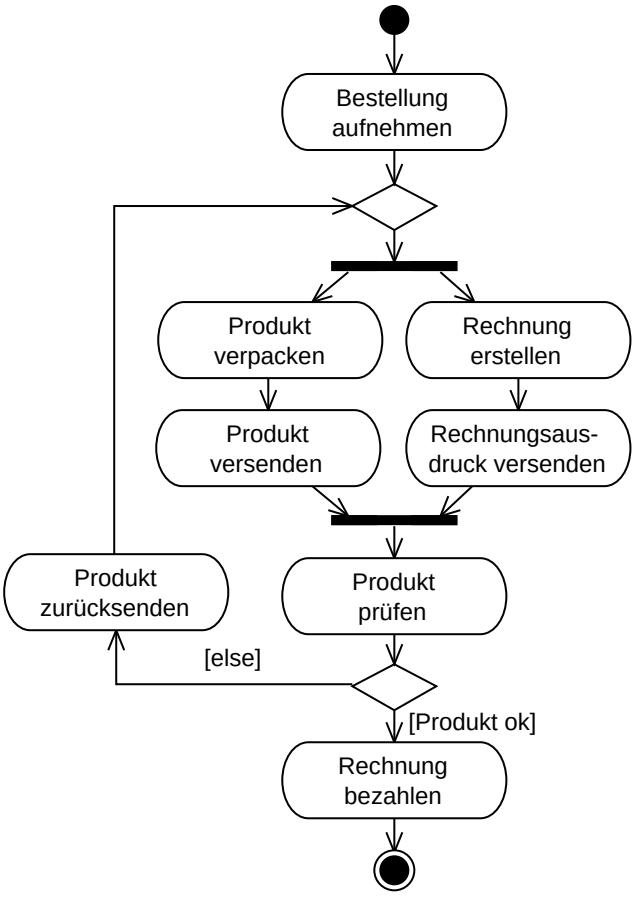

Abbildung 3.2: Bestellung und Auslieferung eines Produkts als Aktivitätsdiagramm.

Abbildung 3.2 zeigt als Beispiel für ein Aktivitätsdiagramm die Bestellung und Auslieferung eines Produkts als Geschäftsprozeß zwischen einem Versandhändler und einem Kunden. Durch Synchronisationsbalken wird modelliert, daß die jeweils in zwei Schritte unterteilten Aufgaben des Produkt- und Rechnungsversands nebenläufig stattfinden können. Das Produkt wird hier erst dann geprüft, wenn es zusammen mit der Rechnung eingetroffen ist. Sodann kann entschieden werden, ob die Rechnung bezahlt oder das Produkt zurückgeschickt wird. Nach dem Bezahlten der Rechnung ist der Ablauf beendet, während in letzterem Fall erneut Produkt und Rechnung versandt werden.

Objektfluß

Die Modellierung von Aktivitäten und Steuerfluß bringt Aufschluß darüber, *was* getan werden muß und *in welcher Reihenfolge*.

Damit ist noch keinerlei Bezug zum restlichen objektorientierten Modell hergestellt. Es werden weitere Konstrukte benötigt, die beschreiben, welche Objekte von Aktivitäten beeinflußt werden oder welche Objekte Aktivitäten beeinflussen. Dazu dient in UML der *Objektfluß*.

Objekte können in Aktivitätsdiagrammen wie in Abschnitt 2.1.1 eingeführt notiert werden. Meist werden keine Attribute angegeben, stattdessen wird das Objekt durch einen in eckigen Klammern notierter *Verarbeitungszustand* näher beschrieben. Dieser sollte dem Zustandsmodell des Objekts entnommen sein.

Für die Diskussion von Kopien in verteilten Systemen interessant ist die mehrfache Darstellung eines Objekts in einem Aktivitätsdiagramm, die hier *Objektvorkommen* genannt wird. Soll durch die mehrfache Notation *dasselbe* Objekt in verschiedenen Verarbeitungszuständen repräsentiert werden, so werden die Vorkommen des Objekts mit gestrichelten Kanten verbunden, die mit dem Stereotyp «becomes» beschriftet werden (siehe dazu auch Abschnitt 2.2.3). Soll dagegen Beschrieben werden, daß ein Objekt eine *Kopie* eines anderen ist, so wird dies durch eine gestrichelte Kante mit dem Stereotyp «copy» ausgedrückt. Objektvorkommen, die nicht mit Kanten verbunden sind, gehören zu verschiedenen Objekten.

Ein Objektvorkommen mit dem ihm zugeordneten Verarbeitungszustand kann Vor- oder Nachbedingung einer Aktivität sein. Eine Aktivität kann weiterhin Objekte erzeugen, wofür es aber in UML keine eindeutig von der Erzeugung eines anderen Verarbeitungszustands zu unterscheidende Notation gibt. In [Hitz und Kappel 1999] wird hierfür der spezielle Verarbeitungszustand [erzeugt] verwendet. Eine Aktivität kann erst gestartet werden, wenn *alle* Vorbedingungs-Objektvorkommen im entsprechenden Verarbeitungszustand vorliegen. Durch entsprechenden Objektfluß können Steuerflußkanten überflüssig werden. Solch redundanter Steuerfluß kann im allgemeinen weggelassen werden, in [Hitz und Kappel 1999], S. 159 wird dies aber bei gleichzeitiger Synchronisation nicht erlaubt, obwohl dies konsequent wäre. Ein Weglassen eines Synchronisationsbalkens bei entsprechenden Objektflüsse würde allerdings dazu führen, daß die Symmetrie von Gabelungen (*forks*) und Vereinigungen (*joins*) nicht mehr direkt erkennbar wäre.

Abbildung 3.2 zeigt einen Ausschnitt aus dem Aktivitätsdiagramm aus Abbildung 3.2, indem zusätzlich Objektflüsse modelliert wurden. Den Verarbeitungszustand notieren wir hier in einem eigenen Abschnitt (*compartment*) und Objektnamen werden nach den Konventionen in dieser Arbeit klein geschrieben. Durch die «becomes»-Kanten läßt sich der Lebenslauf des Produkts während des Ablaufs erkennen. Die Aktivität Rechnungsausdruck versenden wurde so modelliert, daß in dieser die Rechnung neu ausgedruckt und dieser Ausdruck versandt wird, der somit eine Kopie («copy») des Rechnungsobjekts darstellt. Der durch den Objektfluß redundante

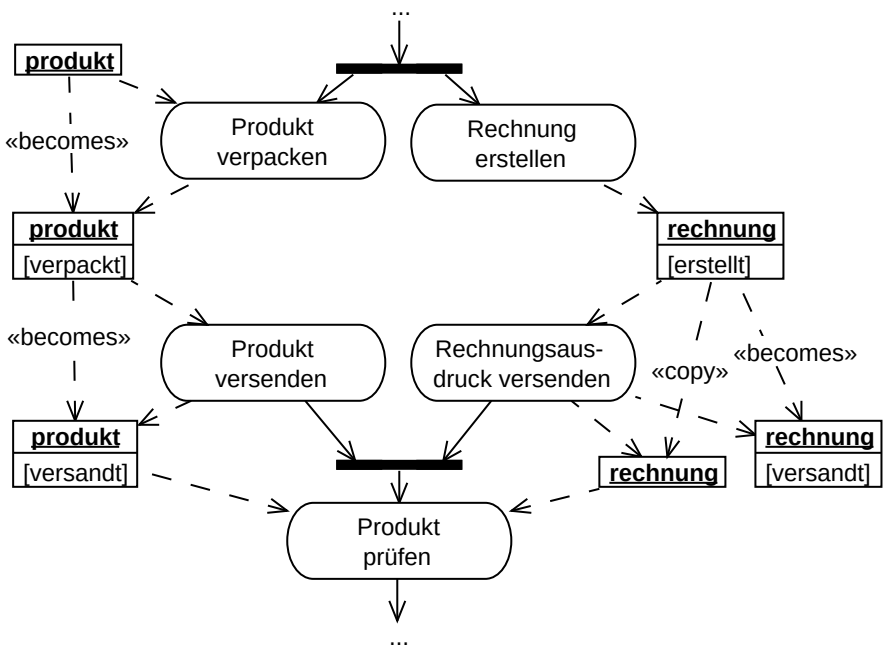

Abbildung 3.3: Ein Ausschnitt aus dem Aktivitätsdiagramm aus Abbildung 3.2 mit Objektflüssen.

Steuerfluß zwischen den einzelnen Schritten der nebenläufigen Aktivitäten wurde wie oben diskutiert weggelassen. Der Synchronisationsbalken zur Vereinigung wurde entsprechend nicht weggelassen, obwohl dies semantisch gleichbedeutend sein sollte.

In Abläufen spielt nicht nur der Zugriff auf Objekte eine Rolle, sondern auch die Zuordnung von Aktivitäten zu Personen ([Graaf und Aalst 1998],[Jablonski et al. 1997]) oder allgemeiner Funktionseinheiten ([Jessen und Valk 1987]). In UML wird dies als *Verantwortlichkeit* für eine Aktivität bezeichnet. Um Verantwortlichkeiten in einem Aktivitätsdiagramm auszudrücken, sieht UML *Verantwortlichkeitsbereiche* (*swimlanes*) vor, denen ein verantwortliches Objekt zugeordnet ist. Jede Aktivität befindet sich in höchstens einem Verantwortlichkeitsbereich. Als verantwortliche Objekte kommen Objekte des zu realisierenden Systems, in Geschäftsprozeßmodellen aber vor allem die entsprechenden Akteure bzw. Rollen in Frage.

Abbildung 3.4 zeigt das Aktivitätsdiagramm aus Abbildung 3.2 mit zusätzlich modellierten Verantwortlichkeitsbereichen. Am oberen Rand werden die Bereiche benannt und durch senkrechte Linien voneinander getrennt. Im Beispiel werden die Aktivitäten dem Versandhändler bzw. dem Kunden zugeordnet, wobei innerhalb des Versandhändler-Unternehmens die Rollen Sachbearbeiter und Aushilfe vorgesehen

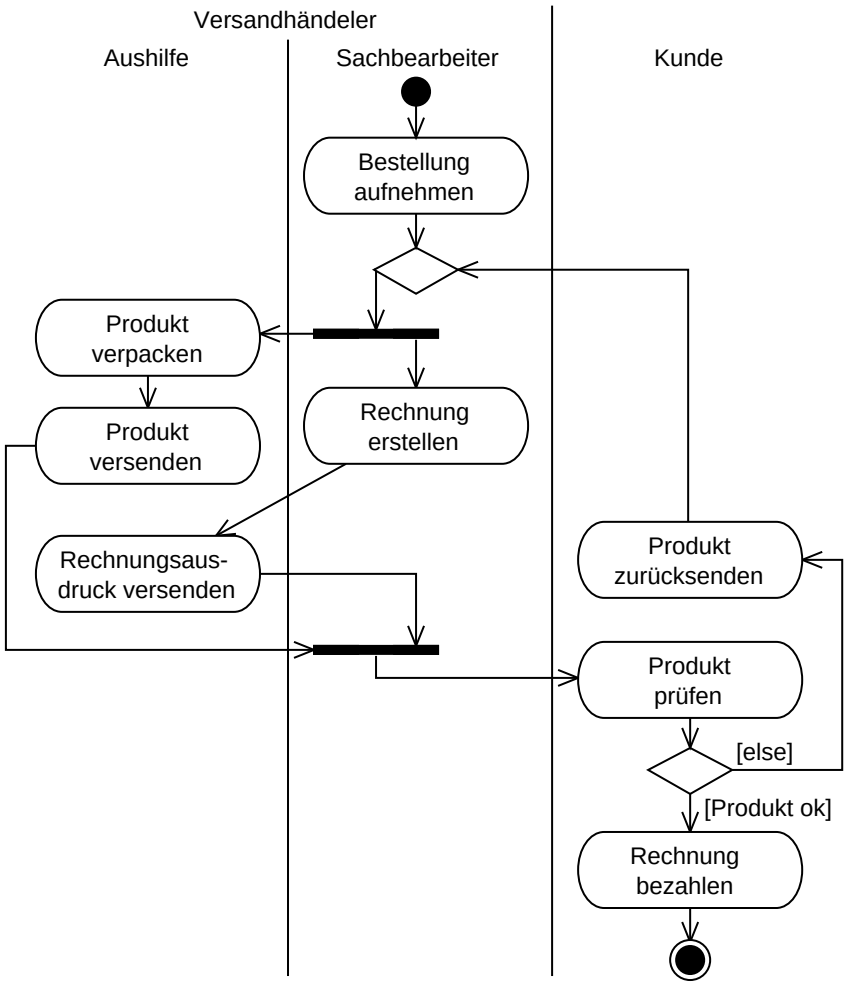

Abbildung 3.4: Das Aktivitätsdiagramm aus Abbildung 3.2 mit Verantwortlichkeitsbereichen.

sind.

Diese Darstellung gibt einen guten Überblick darüber, wer für welche Aufgaben verantwortlich ist. Die Darstellung des Steuerflusses leidet aber unter der Einordnung der Aktivitäten in verschiedene „Schwimmbahnen“. Weiterhin kann eine Aktivität nur genau einem Verantwortlichen zugeordnet werden. Für die Aktivität Bestellung aufnehmen wäre beispielsweise eine Beteiligung des Kunden sinnvoll, auch wenn die Verantwortung dem Sachbearbeiter zufällt.

Zur Modellierung von Geschäftsprozessen wird hier deshalb eine alternative Notation vorgeschlagen, die auf Ideen aus dem Werkzeug „Innovator“ ([MID 2000]) beruht. Ähnlich zum Objektfluß sollen die Akteure innerhalb des Aktivitätsdia-

gramms notiert und über gestrichelte Kanten, die in UML allgemein für Abhängigkeiten (*dependencies*) stehen, mit den Aktivitäten verbunden werden, für die sie verantwortlich sind. Die in UML für Anwendungsfälle bekannte Visualisierung von Akteuren als Piktogramme kann hierbei eingesetzt werden. Für die Abhängigkeiten wird das Stereotyp «performs» vorgeschlagen. Die Vorteile sind, daß

- die zur Darstellung des Steuerflusses gewählte Anordnung der Aktivitäten nicht geändert wird,
- eine Aktivität mehreren Akteuren (oder auch keinem Akteur) zugeordnet werden kann und
- die Art der Zuordnung über Stereotype näher beschrieben werden kann.

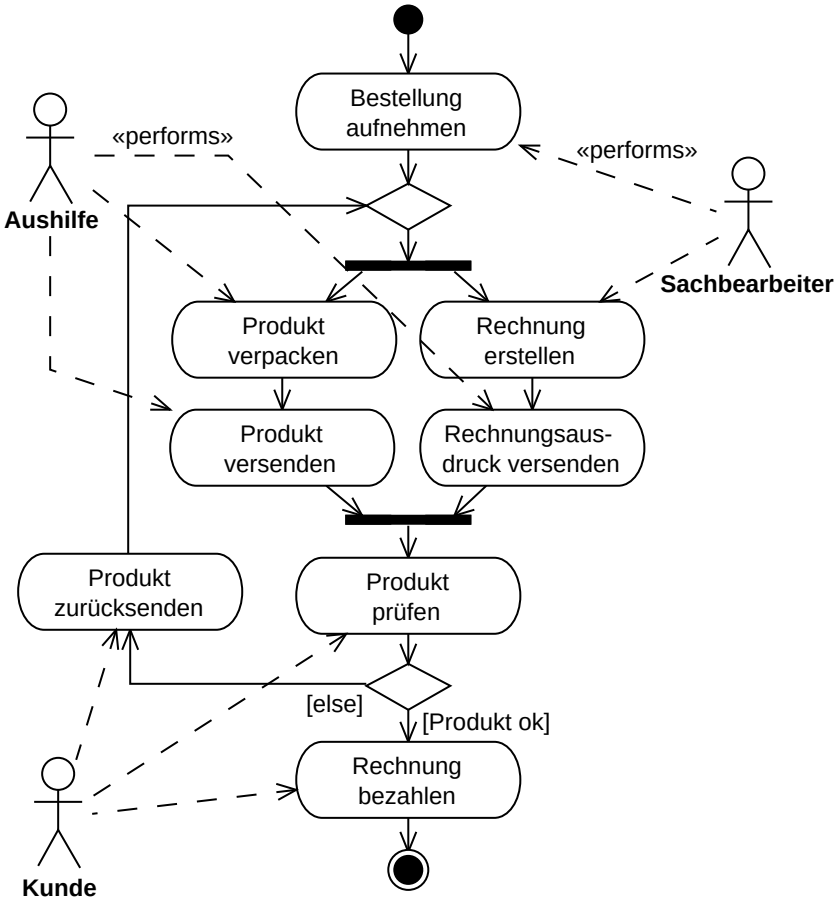

Abbildung 3.5: Das Aktivitätsdiagramm aus Abbildung 3.4 mit Akteuren statt Verantwortlichkeitsbereichen.

Abbildung 3.5 zeigt das Aktivitätsdiagramm aus Abbildung 3.4 in der hier vorgeschlagenen Alternativnotation. Das Stereotyp «performs» ist der Übersichtlichkeit halber nur an zwei Abhängigkeitskanten angedeutet. Man sieht, daß auch diese Darstellung trotz der oben genannten Vorteile schnell unübersichtlich werden kann. Der Grund hierfür liegt darin, daß in einer Modellierung von Aktivitätsdiagrammen mit Verantwortlichkeiten bereits mehrere Sichten gleichzeitig dargestellt werden. Mit einer solchen Vermischung der Sichten muß bei der Modellierung äußert vorsichtig und sparsam umgegangen werden. Für eine genaue Spezifikation, wie sie für ausführbare Modelle nötig ist, führt aber kaum ein Weg daran vorbei, mehreren Sichten des Modells zu betrachten. Sobald die Übersichtlichkeit nachläßt, sollte aber auf eine Darstellung in mehreren Diagrammen ausgewichen werden, die sich auf jeweils eine Sicht konzentrieren.

Vor allem die zuletzt vorgestellte Erweiterung erlaubt einen Einsatz von UML-Aktivitätsdiagrammen zur Modellierung von Geschäftsprozessen und Workflow (siehe Abschnitt 3.4). Diese Einsatzmöglichkeit wird im Anwendungsbeispiel in Abschnitt 5.2.2 wieder aufgegriffen. Nach den UML-Techniken kommen wir nun zu Techniken, welche mit der speziellen Ausrichtung auf die Geschäftsprozeßmodellierung entstanden sind.

3.1.4 Aktivitätenmodelle

Im Aktivitätenmodell, wie es im Werkzeug FlowMark [IBM 1996] eingesetzt wird, gibt es nur eine Art von Knoten, welche die Aktivitäten des Workflows repräsentieren. Kausale Abhängigkeiten und Datenflüsse zwischen den Aktivitäten werden durch zwei verschiedene Kantenarten dargestellt. Eine Aktivität ohne eingehende Kanten wird aktiviert, sobald der Workflow gestartet wird. *Steuerkonnektoren* (durchgezogene Kanten) regeln den Steuerfluß[3], währen Datenkonnektoren (gestrichelte Kanten) die Workflow-Steuer*daten* zwischen den Aktivitäten transportieren.

Abbildung 3.6 aus [Jablonski et al. 1997], S. 167 (vereinfacht) zeigt die Modellierung eines Teils eines Kreditgeschäftsprozesses als Aktivitätenmodell. In diesem Beispiel verlaufen Steuer- und Datenkonnektoren stets parallel. Eine Abweichung von diesem Schema tritt in Fällen auf, in denen eine Entscheidung aufgrund von Steuerdaten getroffen wird, die von einer weiter zurückliegenden Aktivität bestimmt wurden. Da dieser Fall jedoch recht selten vorkommt, wäre eine Kurznotation für parallele Steuer- und Datenkonnektoren wünschenswert.

Steuerkonnektoren können mit einer Bedingung (*Wächter*, *guard*) beschriftet werden, so daß die nachfolgende Aktivität nur dann angesteuert wird, wenn die Bedingung wahr ist. Der Unterschied zwischen Verzweigung und nebenläufiger Bearbeitung mehrerer Aktivitäten ergibt sich in Aktivitätenmodellen daraus, ob bei

[3] In [Jablonski et al. 1997] wird statt Steuerkonnektoren von Kontrollkonnektoren und statt Steuerfluß von Kontrollfluß gesprochen. Der Sprachgebrauch in dieser Arbeit wird in einer Fußnote in der Einleitung begründet.

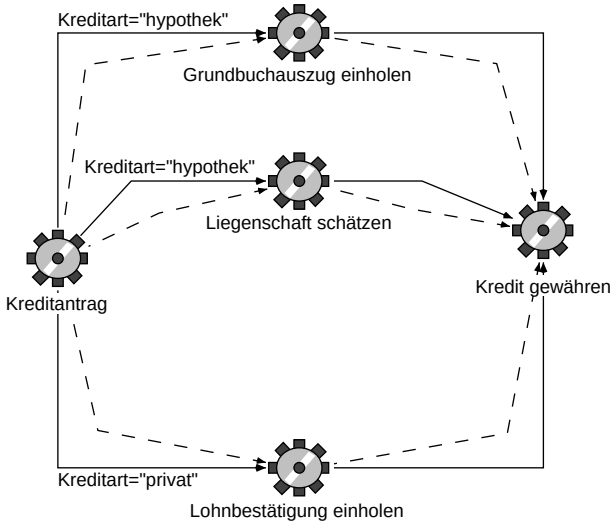

Abbildung 3.6: Aktivitätenmodell eines Kreditbearbeitungs-Workflow nach [Jablonski et al. 1997], S. 167.

mehreren Ausgangskanten einer Aktivität genau eine Bedingung oder mehrere Bedingungen wahr sind. Damit wird die Logik der Steuerung implizit modelliert.

Eine solche implizite Steuerung kommt im Beispiel in Abbildung 3.6 bei Beendigung der initialen Aktivität Kreditantrag vor. Durch die gleichen Wächter mit dem Ausdruck Kreditart="hypothek" an den oberen beiden Kanten werden die Aktivitäten Grundbuchauszug einholen und Liegenschaft schätzen nebenläufig aktiviert, falls eine Hypothek gewünscht wird. Im Fall Kreditart="privat" findet eine Verzweigung zur Aktivität Lohnbestätigung einholen statt. An den Steuerkonnektoren, die in der Aktivität Kredit gewähren zusammenlaufen, ist der semantische Unterschied, ob eines oder mehrere Signale eintreffen müssen, um die Aktivität zu starten, gar nicht zu erkennen.

3.1.5 Ereignisgesteuerte Prozeßketten

Ereignisgesteuerte Prozeßketten (EPK) wurden von Scheer et al. in [Keller et al. 1992, Chen und Scheer 1994] zur Modellierung von Geschäftsprozessen, aber auch als allgemeine Prozeßmodellierungstechnik vorgeschlagen. EPK werden im ARIS-Toolset zur Modellierung von Geschäftsprozessen und in ARIS Workflow und in SAP Business Workflow zur Spezifikation von Workflows verwendet.

Abbildung 3.7 zeigt das Modell des Kreditbearbeitungs-Workflow als EPK, übernommen aus [Jablonski et al. 1997], S. 169, und vereinfacht. Auf den ersten Blick

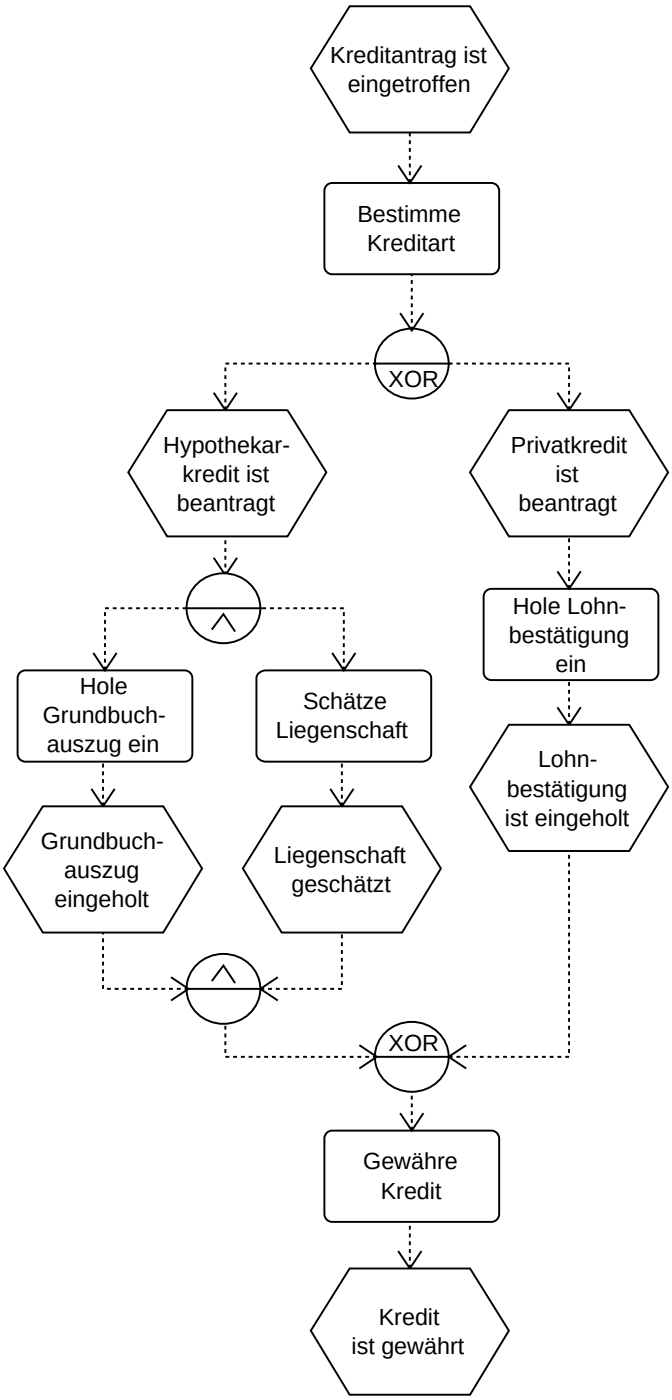

Abbildung 3.7: Ereignisgesteuerte Prozeßkette des Kreditbearbeitungs-Workflow nach [Jablonski et al. 1997], S. 169.

fällt auf, daß sich ein wesentlich umfangreicheres Modell ergibt. Der Grund hierfür ist, daß in EPK Zustände *und* Zustandsübergänge als Knoten modelliert werden und diese demnach als *bipartite* Graphen dargestellt werden. Damit sind EPK prinzipiell Petrinetzen sehr ähnlich, aufgrund bestimmter Entwurfsentscheidungen und einer fehlenden formalen Definition stellte sich die Abbildung auf Petrinetze jedoch als nicht trivial heraus (siehe [Uthmann 1997, Uthmann und Becker 1998, Langner et al. 1998, Aalst 1999b, Moldt und Rodenhagen 2000]).

Zustände werden in EPK als Sechsecke notiert, wobei hier von *Ereignissen* gesprochen wird, die das Eintreten des Zustands auslösen. Diese Terminologie wirkt im Vergleich mit Zustandsdiagrammen (siehe Abschnitt 3.1.2) verwirrend, da dort durch ein Ereignis ein Zustands*übergang* ausgelöst wird, in EPK dagegen der *Zustand*, in dem das Ereignis eingetreten ist, mit dem Ereignis selbst gleichgesetzt wird. Im Beispiel in Abbildung 3.7 sind die Bezeichnungen von „Ereignissen“ wie in der EPK-Modellierung üblich als Zustandsbeschreibungen formuliert. Wir bleiben deshalb beim Begriff *Zustand* für die sechseckigen Knoten.

Funktionen entsprechen den Aktivitäten aus der aktivitätenorientierten Modellierung. Sie können Eingabedaten verarbeiten und Ausgabedaten erzeugen. Funktionen können Zeit verbrauchen und Kosten verursachen. Der Prozeß bleibt dagegen nur dann längere Zeit in einem *Zustand*, wenn durch eine Synchronisation auf die Beendigung einer Funktion gewartet wird. Zustände werden also immer so bald wie möglich verlassen.

Weiterhin werden sogenannte *Verknüpfungsoperatoren* graphisch als Knoten dargestellt, im Modell betrachtet spezifizieren diese jedoch verschiedene Kantenarten. Als Verknüpfungsoperatoren stehen *UND*, *OR* und *XOR* sowie sogenannte Entscheidungstabellen, die hier nicht weiter betrachtet werden sollen, zur Verfügung. Der UND-Operator ($\wedge$) dient der Modellierung von Nebenläufigkeit, während XOR (ausschließendes Oder) einer bedingten Verzweigung entspricht. Der OR-Operator (einschließendes Oder, $\vee$) entspricht der Art der Wächter in der aktivitätenorientierten Modellierung insofern, als je nach zutreffenden Bedingungen eine oder mehrere der folgenden Zweige ausgeführt werden können. Vor allem dieser Operator stellt sich in [Langner et al. 1998, Aalst 1999b, Moldt und Rodenhagen 2000] als schwierig formalisierbar heraus, da seine Semantik unklar ist. Da die beiden anderen Operatoren die grundlegenden Modellierungsmittel zur Verfügung stellen, wird hier von einer Verwendung des OR-Operators abgeraten und dieser nicht weiter betrachtet.

Die oben erwähnten Redundanzen durch Zustands- und Aktivitätsknoten sind im EPK in Abbildung 3.7 deutlich zu erkennen. Ein EPK-Prozeßmodell beginnt und endet mit Zuständen. Am Beispiel der Kreditart sieht man, wie in EPK eine durch unterschiedliche Ereignisse ausgelöste Entscheidung modelliert wird (oberer XOR-Operator). Im Unterschied zu Aktivitätenmodellen werden auch beim Zusammenführen alternativer oder paralleler Zweige des Prozesses verschiedene Kantenarten unterschieden, indem ein Hilfsknoten mit der Operatorbezeichnung in der oberen Hälfte notiert wird (z.B. unterer XOR-Operator). Als abkürzen-

de Notation können mehrere Anfangs- und Endzustände angegeben werden. Ob diese alternativ oder gleichzeitig eingenommen werden sollen, kann aber nur an der Behandlung im weiteren Prozeß erkannt werden. Diese Abkürzung kann dadurch leicht zu Modellierungsfehlern führen und ist auch nicht immer eindeutig ([Aalst 1999b, Moldt und Rodenhagen 2000]).

Da EPK abgesehen von den semantischen Unklarheiten Petrinetzen stark ähneln, werden in dieser Arbeit Petrinetze als Prozeßmodellierungstechnik eingesetzt. Auf Anwendungen von Petrinetzen in der Geschäftsprozeßmodellierung geht Abschnitt 3.4 ein. In den nächsten Abschnitt werden zuvor Grundlagen für Petrinetze und ausgewählte erweiterte Petrinetz-Formalismen dargestellt.

3.2 Klassische Petrinetze

In diesem Abschnitt werden Petrinetze als allgemeines Beschreibungsmittel für Kausalität und Nebenläufigkeit eingeführt. Man unterscheidet verschiedene Arten von Petrinetzen, die z.B. in [Reisig 1986, Jessen und Valk 1987, Brauer et al. 1987, Baumgarten 1990] und in [Jensen 1992] vorgestellt werden. Hier soll ein Überblick über die für diese Arbeit relevanten Typen von Petrinetzen gegeben werden. Für Definitionen wird auf Anhang B verwiesen.

3.2.1 Basis-Petrinetze

Bevor S/T-Netze und gefärbte Petrinetze eingeführt werden, soll die Basisterminologie der Netztheorie an den grundlegenden Varianten Auftragssystem, Kausalnetz und B/E-Netz dargestellt werden.

Auftragssysteme

Bei dieser sehr einfachen Variante handelt es sich genaugenommen nicht um Petrinetze. Auftragssysteme bestehen nur aus einer Knotenart, den Aufträgen, die durch Kanten verbunden werden, die eine Präzedenzrelation zwischen den Aufträgen festlegen. Diese Präzedenzrelation muß sinnvollerweise irreflexiv und transitiv sein und definiert somit eine partielle Ordnung auf den Aufträgen (siehe Definition A.4). Durch die Beschränkung auf eine Knotenart sind Auftragssysteme einfach zu verstehen und zu notieren. Auftragssysteme werden im folgenden in Form von Kausalnetzen als ein Spezialfall von Petrinetzen beschrieben. Dies ermöglicht einheitliche Definitionen für Eigenschaften von Netzen und für deren sogenannte *operationale Semantik*, die einer statischen Netzstruktur eine exakt definiertes *Verhalten* zuweist.

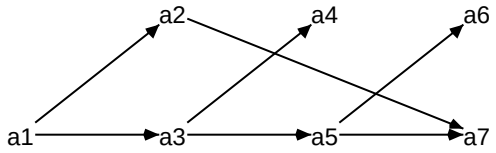

Abbildung 3.8: Konnektivitätsgraph eines Auftragssystems aus [Jessen und Valk 1987], S. 25.

Kausalnetze

Ein Kausalnetz ist ein zyklusfreies Petrinetz, in dem jede Stelle höchstens eine Eingangs- und und Ausgangstransition hat. Kausalnetze stellen den einfachsten Petrinetzformalismus dar, der Auftragssysteme abzubilden vermag. [Jessen und Valk 1987] gibt an, wie zu einem gegebenen Auftragssystem das entsprechende *zugeordnete Kausalnetz* erzeugt und umgekehrt aus einem Kausalnetz das *zugeordnete Auftragssystem* abgeleitet werden kann.

Abbildung 3.9 zeigt das dem Auftragssystem aus Abbildung 3.8 zugeordnete Kausalnetz aus [Jessen und Valk 1987], S. 26. Die Stellennamen sind willkürlich gewählt, wogegen die Namen der Transitionen denen der Aufträge entsprechen.

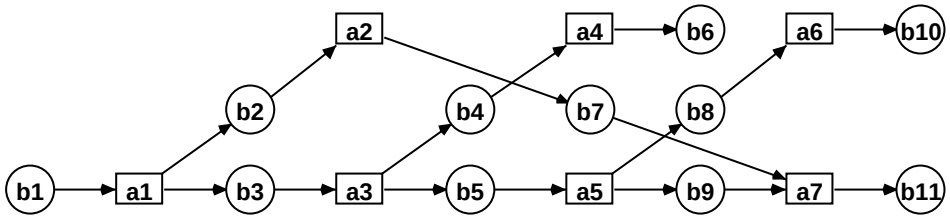

Abbildung 3.9: Dem Auftragssystem aus Abbildung 3.8 zugeordnetes Kausalnetz aus [Jessen und Valk 1987], S. 26.

Kausalnetze weisen die Eigenschaft auf, keinerlei Konfliktstellen zu enthalten, wobei eine Konfliktstelle eine Stelle mit mehr als eine Ausgangstransition ist. Diese Eigenschaft ist für die Darstellung von Prozessen wichtig, weil diese ebenfalls keine Konfliktstellen enthalten.

Kausalnetzen kann eine operationale Semantik gegeben werden, wenn man Markierungen einführt. Man spricht hier von markierten Kausalnetzen, die einen Sonderfall von B/E-Netzen darstellen.

Die Definition von Kausalnetzen ist im Anhang in Definition B.4 wiedergegeben.

B/E-Netze

Das Bedingungs/Ereignis- oder auch B/E-Netz-Modell betrachtet Petrinetze, in denen in jeder Stelle höchstens eine Marke liegen darf. Damit verhalten sich die Stellen wie Wahrheitswerte (markiert oder nicht markiert) und werden entsprechend *Bedingungen* genannt. Bei Transitionen, welche die Bedingungen ändern, spricht man von *Ereignissen*. Die Eingangsstellen eines Ereignisses heißen auch *Vorbedingungen*, die Ausgangsstellen *Nachbedingungen*. Stellen, die Vor- und Nachbedingung sind, werden als *Nebenbedingungen* bezeichnet.

Während Netze zunächst nur als Strukturen betrachtet werden, die kausale Abhängigkeiten angeben, können durch die *Markierung* von Netzen (siehe Definition B.5) verschiedene aktuelle Zustände unterschieden werden. Eine markierte Stelle wird graphisch durch einen ausgefüllten Kreis in der Stelle dargestellt. Allgemein werden markierte Petrinetze bzw. Petrinetze zusammen mit einer Markierung auch als Netz*systeme* bezeichnet. Aufgrund ihrer einfachen Struktur werden B/E-Netzsysteme auch *elementare Netzsysteme* (*elementary net systems*) genannt.

Da B/E-Netze einen Sonderfall der im nächsten Abschnitt vorgestellten S/T-Netze darstellen, ist für diese eine operationale Semantik definiert, die auch als Markenspiel (*token game*) bezeichnet wird. Netzsysteme besitzen eine *Anfangs-* oder *Initialmarkierung*, welche die Markierung zu Beginn des Markenspiels spezifiziert.

Die formale Definition eines B/E-Netzes ist im Anhang in Definition B.7 angegeben.

3.2.2 S/T-Netze

S/T-Netze sind das am häufigsten anzutreffende Petrinetz-Modell, da sie einen Kompromiß zwischen theoretischer Beherrschbarkeit und Ausdrucksmächtigkeit darstellen. Während einerseits B/E-Netze zur Modellierung vieler praktischer Probleme nicht ausreichen oder sehr umfangreiche Modelle erzeugen, sind für die zur praxisnahen Modellierung geeigneten gefärbten Netze (siehe nächster Abschnitt) viele interessante Problemstellungen nicht mehr effizient oder gar nicht entscheidbar. Beispiele für solche Probleme sind das Erreichbarkeitsproblem (Kann eine bestimmte Markierung durch ein gegebenes Netz erreicht werden?) und die Berechnung von Invarianten, die sich nur eingeschränkt für gefärbte Netze realisieren läßt ([Valk 1993, Reisig 1990]).

Man kann ein S/T-Netz als eine Vergröberung oder auch *Faltung* eines B/E-Netzes einführen, wie als Beispiel in Abbildung 3.10 dargestellt. Mehrere Bedingungen werden zu einer Stelle zusammengefaßt, so daß je nach *Anzahl* der wahren Bedingungen mehrere Marken auf der Stelle liegen können. Diese werden graphisch entweder als viele ausgefüllte Kreise (*black tokens*) oder einfach als die natürliche Zahl der Marken in der Stelle dargestellt. Die maximale Anzahl, die sich aus der Anzahl der Bedingungen ergibt, die zusammengefaßt wurden, wird *Kapazität* der

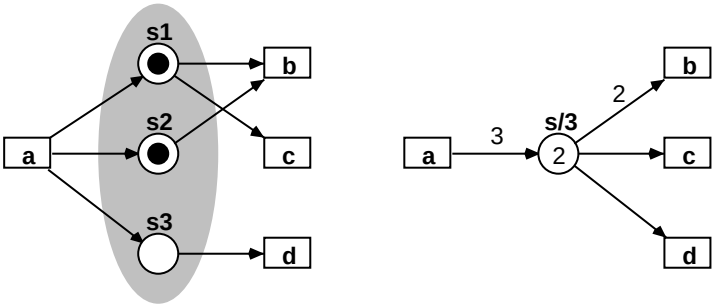

Abbildung 3.10: Vergröberung bzw. Faltung eines B/E-Netzes zu einem S/T-Netz mit Kapazitäten und Kantengewichten aus [Jessen und Valk 1987], S. 35.

Stelle genannt. Bei der Vereinigung von Bedingungen zu Stellen werden die Kanten des B/E-Netzes in das S/T-Netz abgebildet. Da es hier zu Mehrfachkanten kommen kann (also mehrere Kanten, die in derselben Richtung von derselben Transition zu derselben Stelle führen oder umgekehrt), wird als Kurznotation eingeführt, daß Kanten mit einer natürlichen Zahl als *Kantengewicht* beschriftet werden können. Diese abkürzende Schreibweise tritt auch in der Definition von S/T-Netzen auf, wo sie sich als *Kantenbewertung* oder auch *Kantengewichtsfunktion* wiederfindet.

S/T-Netze können durch B/E-Netze dargestellt werden, solange alle Stellen endliche Kapazität aufweisen. Wird aber eine unendliche Kapazität erlaubt, so ergäben sich unendliche B/E-Netz-Strukturen, die in der Definition explizit ausgeschlossen werden. Erst dann ist das S/T-Netz also eine echte Erweiterung des B/E-Netzes.

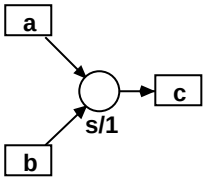

Abbildung 3.11: Beispiel für eine Kapazitätsbeschränkung, die zu einem Konflikt im Nachbereich führt.

Wenn ohnehin unendliche Kapazitäten erlaubt sein sollen, so liegt es nahe, dies als Standard für Stellen anzunehmen. Endliche Kapazitäten führen nämlich zu einigen in bestimmten Situationen unerwünschten Eigenschaften: Eine Konfliktstelle (siehe oben) kann sich nicht nur dadurch ergeben, daß mehrere Transitionen dieselbe Eingangsstelle teilen, sondern auch durch geteilte Ausgangsstellen, wenn das Schalten beider Transitionen nur deswegen nicht möglich ist, weil die Kapazität einer der

Ausgangsstellen überschritten werden würde. Man spricht hier auch von einem Konflikt im Nachbereich. Ein Beispiel dafür zeigt Abbildung 3.11. Kapazitäten führen hier durch kontraintuitive Konflikte zur Einschränkung von Nebenläufigkeit.

Um erkennen zu können, in welchen Fällen diese unerwünschten Eigenschaften auftreten, wird der Begriff der *Kontaktfreiheit* eingeführt: Ein S/T-Netz heißt genau dann kontaktfrei, wenn eine ausreichende Anzahl von Marken im Vorbereich einer Transition eine hinreichende Bedingung[4] für deren Aktivierung ist. Damit werden Konflikte im Nachbereich ausgeschlossen. Ein S/T-Netz ohne Kapazitäten ist demnach immer kontaktfrei.

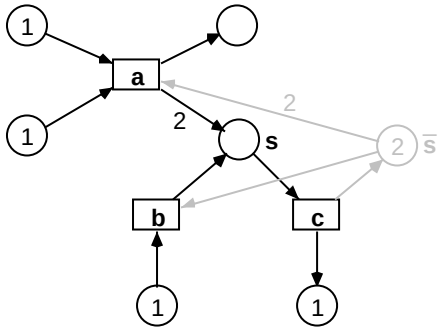

Abbildung 3.12: Konstruktion einer Komplementärstelle zu einer Stelle mit Kapazitätsbeschränkung aus [Jessen und Valk 1987], S. 51.

Die Wirkung von Kapazitäten kann durch die Konstruktion von *Komplementärstellen* in S/T-Netzen ohne Kapazitätsbeschränkungen simuliert werden. Dazu wird zu jeder Stelle s, die eine Kapazität k aufweisen soll, eine neue Stelle $\bar{s}$ eingeführt, die als Anfangsmarkierung $k - M_0(s)$ Marken enthält. Jede Transition muß genau so viele Marken in $\bar{s}$ ablegen, wie sie aus s entnimmt, und so viele Marken aus $\bar{s}$ entnehmen, wie sie in s ablegt. Dadurch entsteht die Stelleninvariante (siehe z.B. [Baumgarten 1990]) $m(s) + m(\bar{s}) = k$, die impliziert, daß die Markierung von s nie über k steigen kann. Abbildung 3.12 aus [Jessen und Valk 1987], S. 51, zeigt ein einfaches Beispiel für diese Konstruktion. Die Stelle s besitzt die Kapazität, die durch die grau dargestellte Komplementärstelle s̄ und die Kanten sichergestellt wird. Daß tatsächlich noch dieselben Schaltfolgen wie mit direkter Berücksichtigung der Kapazität möglich sind, wird in [Jessen und Valk 1987] für den allgemeinen Fall gezeigt.

Im vorherigen Abschnitt wurde auf das Markenspiel hingewiesen, das Petrinetzen eine Ausführungssemantik verleiht. Definition B.5 legt den Begriff der *Markierung* fest, der jeder Stelle eine Anzahl von Marken zuweist. In Definition B.6 werden

[4] Eine ausreichende Anzahl von Marken im Vorbereich einer Transition ist in jedem Fall eine *notwendige* Bedingung für die Aktivierung der Transition.

S/T-Netze formal definiert. Beides stellt eine Grundlage für Definition B.8 dar, die *Aktivierung* und *Schalten* von Transitionen in S/T-Netzen formalisiert.

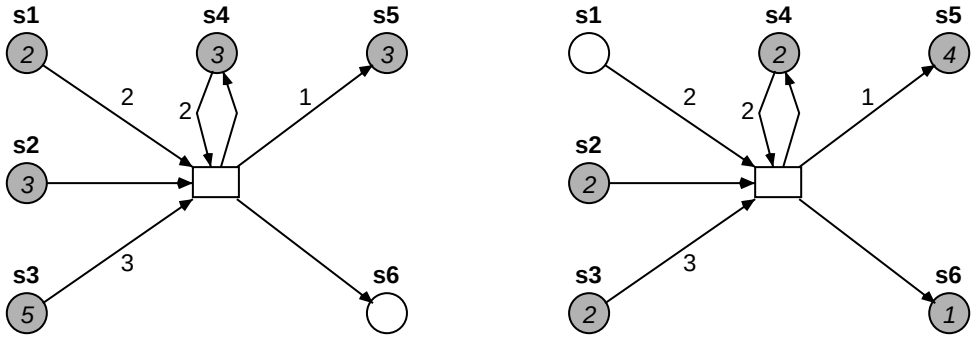

Abbildung 3.13: Ein markiertes S/T-Netz ohne Kapazitäten aus [Jessen und Valk 1987], S. 35. Die aktivierte Transition (links) schaltet in eine Folgemarkierung (rechts).

Abbildung 3.13 zeigt ein Beispiel für ein markiertes S/T-Netz ohne Kapazitäten aus [Jessen und Valk 1987], S. 35. Die aktuelle Markierung wird durch die natürlichen Zahlen innerhalb der Stellen angegeben. Informal beschrieben ist eine Transition in einem S/T-Netz *aktiviert*, wenn auf ihren Eingangsstellen mindestens die aus den Kantengewichten hervorgehende Anzahl von Marken vorhanden ist. Da diese Bedingung in der Abbildung links erfüllt ist, ist die Transition aktiviert. Die Transition *schaltet* und erzeugt dabei eine neue Markierung, indem sie diese Marken aus den Eingangsstellen entfernt und durch die Kantengewichte der Ausgangskanten festgelegte Anzahlen von Marken auf den Ausgangsstellen ablegt. Die rechte Seite der Abbildung zeigt die Folgemarkierung nach dem Schalten.

Schaltfolgensemantik und Erreichbarkeit

Durch wiederholtes Anwenden der Schaltregel entstehen *Schaltfolgen*, die als Folgen von Transitionen dargestellt werden können. Eine Markierung heißt *erreichbar*, wenn sie aus der Anfangsmarkierung durch eine Schaltfolge hergestellt werden kann. Da in einer Markierung eines Netzes mehrere Transitionen aktiviert sein können, enthält die Ausführungssemantik von S/T-Netzen Nichtdeterminismus.

Abbildung 3.14 zeigt als ein inhaltlich motiviertes Beispiel für S/T-Netze das bekannte Problem des wechselseitigen Ausschlusses von Lesern und Schreibern aus [Jessen und Valk 1987], S. 39 (dort mit drei Marken dargestellt, siehe auch Abschnitt 1.1.1). In der Stelle lok liegen zwei Marken, die Prozesse (nicht zu verwechseln mit Petrinetz-Prozessen) repräsentieren, die auf einen kritischen Abschnitt lesend oder schreibend zugreifen. Der linke Bereich repräsentiert lesende, der rechte schreibende Prozesse. Beide Vorgänge werden in zwei Schritten modelliert: Zunächst wird der Wunsch zu lesen oder zu schreiben gemeldet (Transitionen a bzw. d und

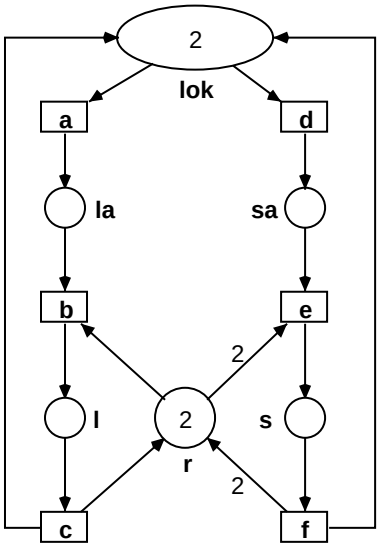

Abbildung 3.14: Wechselseitiger Ausschluß von Schreibern und Lesern mit zwei Aufträgen (in [Jessen und Valk 1987], S. 39, mit drei Aufträgen).

Stellen la bzw. sa). Dann wird der kritische Abschnitt im jeweiligen Modus betreten (Transitionen b bzw. e und Stellen l bzw. s). Um die Konsistenz der Daten im kritischen Abschnitt zu gewährleisten, darf nur ein Prozeß zu Zeit schreiben, lesen dürfen aber beliebig viele. Dies modelliert die Stelle r, aus der jeder Leser eine Marke entnehmen muß, jeder Schreiber hingegen alle Marken. Damit ist sichergestellt, daß ein Schreiber nur dann den kritischen Abschnitt betreten kann, wenn sich kein Leser darin befindet. Wird der kritische Abschnitt verlassen (Transition c bzw. f), müssen alle Marken zurückgelegt werden.

Um das mögliche Schaltverhalten und alle erreichbaren Markierungen im Überblick darzustellen, wird der *Erreichbarkeitsgraph* eines S/T-Netzes definiert. Dieser kann allerdings unendlich groß sein, so daß er sich nicht immer effektiv erstellen läßt. In diesem Fall kann ein *Überdeckungsgraph* konstruiert werden, der unendliche Mengen von Markierungen zusammenfaßt. Mithilfe des Überdeckungsgraphen läßt sich beispielsweise feststellen, welche Stellen eine beliebig große Anzahl von Marken akkumulieren können (siehe [Jessen und Valk 1987]).

Bereits für das einfache Beispiel aus Abbildung 3.14 entsteht der relativ komplexe Erreichbarkeitsgraph in Abbildung 3.15. Die Markierungen werden als Sequenzen von Stellennamen dargestellt. Eine mehrfache Markierung wird als Faktor mit einem Apostroph vorangestellt.

Der Erreichbarkeitsgraph besitzt einen Wurzelknoten, welcher der Initialmarkierung des Netzes entspricht und graphisch durch eine eingehende Kante ohne

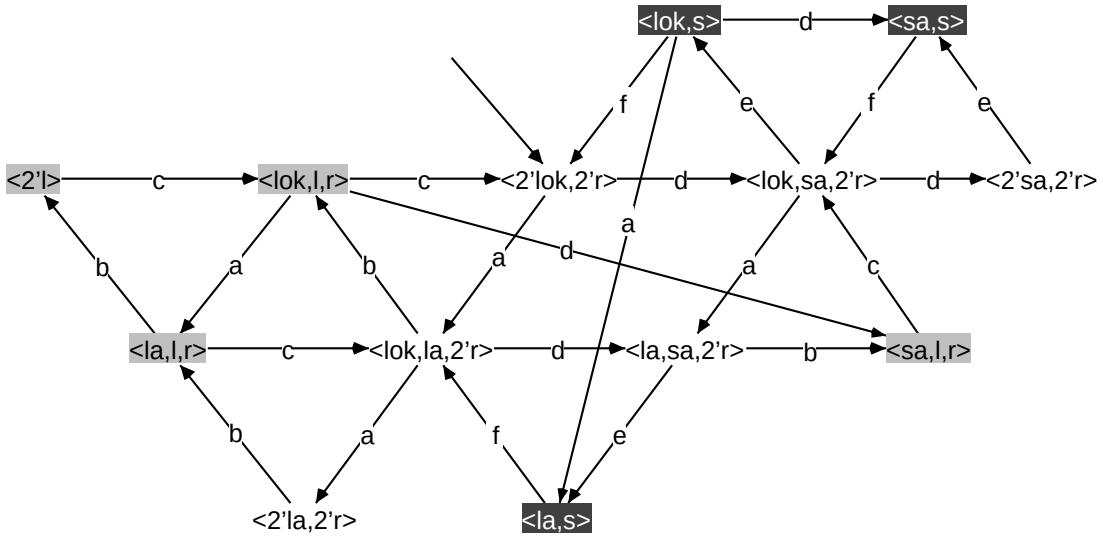

Abbildung 3.15: Der Erreichbarkeitsgraph des S/T-Netzes aus Abbildung 3.14.

Startknoten als Wurzelknoten ausgezeichnet wird. Von jedem Knoten des Erreichbarkeitsgraphen geht für jede in der dargestellten Markierung aktivierte Transition eine Kante aus, die auf den Knoten zeigt, der die Folgemarkierung darstellt. Pfade durch den Erreichbarkeitsgraphen sind mögliche Schaltfolgen des Netzes. Die durch die Schaltfolge erreichte Markierung läßt sich direkt aus dem Zielknoten ablesen, den man über den Pfad erreicht.

Hervorgehoben sind im Beispiel in Abbildung 3.14 die Zustände, in denen sich mindestens ein Leser (hellgrau) bzw. genau ein Schreiber (dunkelgrau) im kritischen Abschnitt befindet. Gut zu erkennen sind die *Transitionsinvarianten*, also (Teil-)Schaltfolgen, die eine zuvor erreichte Markierung reproduzieren. Dies sind insbesondere Zyklen der Transitionen `a`, `b`, `c` des linken bzw. `d`, `e`, `f` des rechten Bereichs. Auch Stelleninvarianten (siehe oben) sind im Erreichbarkeitsgraphen erkennen, da dieser alle erreichbaren Markierungen enthält (die Kanten werden hier nicht benötigt).

Schaltfolgen und Erreichbarkeitsgraphen beschreiben das Verhalten eines S/T-Netzes in einer *Schaltfolgensemantik*, in der jeweils das Schalten einer einzelnen Transition zur Zeit betrachtet wird. Es gibt aber Transitionen, deren Schalten in bestimmten Markierungen unabhängig voneinander stattfinden kann. Um dieses Verhalten bei der Ausführung angemessen zu berücksichtigen, gibt es zwei andere Semantiken: Die *Schrittsemantik* und die *Prozeßsemantik*.

Schrittsemantik

Die Schrittsemantik stellt das gleichzeitige Schalten einer Multimenge von unabhängigen Transitionen als einen *Schritt* dar. Multimengen (siehe Definition A.11) werden hier verwendet, da dieselbe Transition in einem S/T-Netz bei ausreichender Markierung in einem Schritt mehrfach auftreten kann. In B/E-Netzen sind Transitionen unabhängig, wenn sie einen disjunkten Vor- und Nachbereich aufweisen. In S/T-Netzen muß mindestens die Summe aller von den Transitionen des Schritts konsumierten Marken in den entsprechenden Stellen vorhanden sein. Alle Transitionen eines aktivierten Schritts können gleichzeitig schalten. Dafür wird eine entsprechende Schaltregel für Schritte definiert (siehe Anhang, Definition B.9).

Schritte sind im Beispiel aus Abbildung 3.14 bei entsprechender Markierung für alle zweielementigen Transitionsmultimengen möglich, die nicht e oder f enthalten. Dadurch, daß e alle Marken aus r abzieht, kann nie eine andere Transition gleichzeitig zu e schalten. f ist von e kausal abhängig. Da e immer nur einmal schalten kann, bis f geschaltet hat, kann auch f nicht gleichzeitig mit einer anderen Transition schalten.

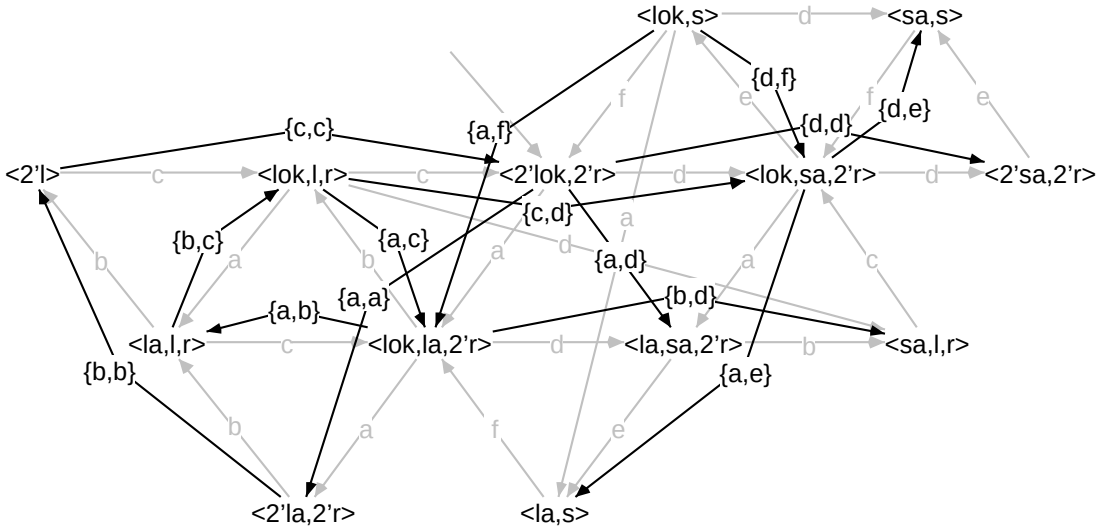

Abbildung 3.16: Der Erreichbarkeitsgraph des S/T-Netzes aus Abbildung 3.14 mit zusätzlichen Kanten für Schritte.

Eine notwendige Bedingung für die Aktivierung eines Schrittes ist, daß alle Permutationen der Multimengenelemente als Schaltfolgen aktiviert sind. Daß diese Bedingung nicht auch hinreichend ist, mache man sich am Beispiel eines Netzes klar, das alle möglichen Permutationen über einer Menge von Transitionen als alternative Sequenzen erlaubt.

Schritte können auch im Erreichbarkeitsgraphen dargestellt werden. Sie führen

zu zusätzlichen Kanten, die mit den Transitionsmultimengen beschriftet werden. Nach der oben angegebenen Regel bestimmt sich der Zielknoten einer Schrittkante, indem für eine beliebige Reihenfolge der Multimengenelemente der Zielknoten der Schaltfolge bestimmt wird. Insofern stellen Schrittkanten „Abkürzungen“ im Erreichbarkeitsgraphen dar. Abbildung 3.16 zeigt als Erweiterung zu Abbildung 3.15 die im Beispiel hinzukommenden Kanten.

Prozeßsemantik

Um die Unabhängigkeit und damit Nebenläufigkeit von Transitionen noch stärker zu betonen, werden Abläufe von Petrinetzen als *Prozesse* dargestellt. Im Beispiel aus Abbildung 3.14 tritt Nebenläufigkeit beispielsweise in der Anfangsmarkierung auf, in der die Transitionen a und d aktiviert sind. In der Schaltfolgensemantik wird nichtdeterministisch eine der beiden Transition ausgewählt, die zuerst schaltet. Die andere Transition bleibt aktiviert und kann später schalten. In der Schrittsemantik ist zusätzlich das gleichzeitige Schalten von a und d möglich.

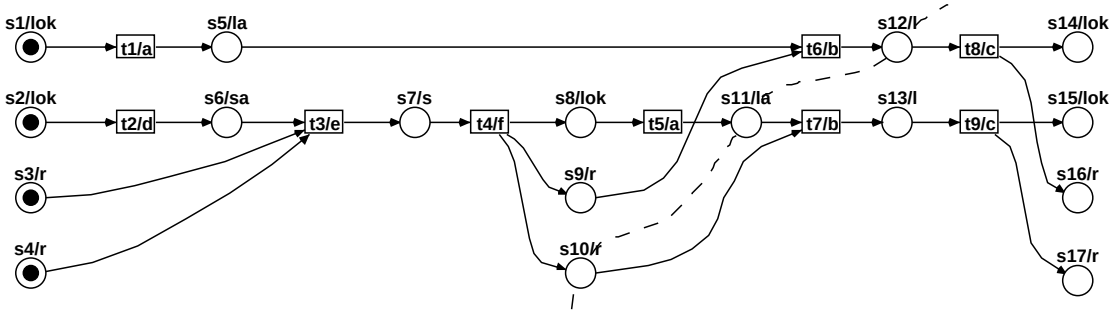

Abbildung 3.17: Ein Prozeß des S/T-Netzes aus Abbildung 3.14 leicht modifiziert aus [Jessen und Valk 1987], S. 42.

Abbildung 3.17 zeigt einen asynchronen Prozeß des Netzes aus Abbildung 3.14 und basiert auf [Jessen und Valk 1987], S. 42. Da in dieser Arbeit nur asynchrone Prozesse betrachtet werden, wird dieser Zusatz im folgenden weggelassen. Ein Prozeß eines S/T-Netzes wird wiederum als Petrinetz dargestellt. Alle Stellen und Transitionen des Prozeßnetzes sind entsprechenden Netzelementen des Basisnetzes zugeordnet. Dies ist in der Abbildung durch die Namensvergabe gekennzeichnet: dem Namen des Netzelements im Prozeßnetz folgt hinter einem Schrägstrich der Name des entsprechenden Elements im Basisnetz. Man spricht hier von einem *Vorkommen* (*occurrence*) des Netzelements im Prozeß. Jede Transition eines Prozesses eines Netzes N stellt ein Transitionsvorkommen (*transition occurrence*) einer Transition aus N dar, gleiches gilt für ein Stellenauftreten (*place occurrence*) im Prozeß.

Im Prozeß sind alle Konflikte (siehe Abschnitt 3.2.2) aufgelöst, da die Entschei-

dung, welche der im Konflikt stehenden Transitionen schaltet, hier bereits getroffen wurde. Darum gibt es zu einem Netz mit Konflikten im allgemeinen mehrere (auch unendlich viele) unterschiedliche Prozesse. Es soll nochmals betont werden, daß unterschiedliche Prozesse nur durch unterschiedlich gelöste Konflikten entstehen, gerade nicht durch die Auswahl, um welche von mehreren unabhängig voneinander aktivierten Transitionen der Prozeß zuerst[5] erweitert wird. Durch die Auflösung von Konflikten und die Auffaltung von mehrfach markierten Stellen zu mehreren einfach markierten Stellen erhalten Prozeßnetze zu S/T-Netzen die Struktur eines *markierten Kausalnetzes*.

Erst die Prozeßsemantik ermöglicht bei Petrinetzen die Betrachtung von Nebenläufigkeit während der Ausführung. In Prozessen lassen sich kausale Abhängigkeiten der Vorkommen von Stellen und Transitionen ablesen, wenn es eine gerichtete Verbindung von einem Element zum anderen gibt. Man sagt, die Elemente befinden sich *in Linie* (*in line*, Relation li in [Jessen und Valk 1987], S. 27). Nicht kausal abhängige Elemente heißen nebenläufig (Relation co in [Jessen und Valk 1987], S. 27). Maximale Kliquen von nebenläufigen Elementen werden als *Schnitte* (*cuts*) bezeichnet, wobei aus Schnitten von Stellen, sogenannten *S-Schnitten* (*S-cuts*), mögliche Markierungen des Netzes, aus dem der Prozeß hervorgegangen ist, abgelesen werden können ([Jessen und Valk 1987]). In Abbildung 3.17 ist ein S-Schnitt durch eine gestrichelte Linie dargestellt.

3.2.3 Höhere Petrinetze

Es ist allgemein anerkannt, daß Petrinetze sich zwar sehr gut zur Modellierung von Kausalität und Nebenläufigkeit und damit zur Modellierung von Prozessen eignen, daß aber in Prozessen auch die Abfrage und Bearbeitung von Daten modelliert werden muß ([Jensen 1992]). Ein Ansatz ist, Petrinetze nur zur Modellierung der Kontrollstrukturen zu nutzen und die Datenmanipulation und -abfrage beispielsweise in einer Programmiersprache zu formulieren. In [Jessen und Valk 1987] werden solche Darstellungen *Netzprogramme* genannt.

Um Daten und Prozesse aber noch enger miteinander in Verbindung zu bringen, wurden Petrinetze so weiterentwickelt, daß die in S/T-Netzen anonymen Marken Daten enthalten können. Solche Petrinetze werden im allgemeinen unter dem Begriff „höhere Petrinetze“ zusammengefaßt.

Um die formale Genauigkeit der Petrinetze auch bei der Beschreibung der durch Marken repräsentierten Daten zu erhalten, wurden verschiedene Ansätze vorgeschlagen.

Prädikaten/Transitions-Netze (Pr/T-Netze, [Genrich 1987]) bedienen sich der klassischen Prädikatenlogik, um Marken und Transitionsanschriften zu spezifizieren. Während die Marken Grundterme (*ground terms*) sind, werden für die Transitionen

[5] „Zuerst“ bezieht sich hier nicht auf die kausale Reihenfolge von Transitionen im Prozeß, sondern auf die Reihenfolge, in welcher der Prozeß *konstruiert* wird.

prädikatenlogische Formeln verwendet. Kanten werden mit Variablen der Formeln oder mit Konstanten beschriftet. Eine Transition ist aktiviert, wenn es für die Variablen der Formel eine zulässige Belegung gibt, so daß Marken mit entsprechenden Termen, die den Belegungen der Variablen an den Kanten gleichen, in den Eingangsstellen vorhanden sind. Aus der Variablenbelegung beim Schalten ergibt sich dann der Wert der abzulegenden Marken.

In algebraischen Netze ([Reisig 1990]) werden Marken durch algebraisch definierte abstrakte Datentypen beschrieben. Durch die auf abstrakten Datentypen gegebenen Operationen können auch die Transitionswächter (siehe unten) und -funktionen mathematisch exakt beschrieben werden.

In der Modellierung realer Systeme wird vor allem eine Art der höheren Petrinetze eingesetzt, die gefärbten Petrinetze nach Jensen.

Gefärbte Petrinetze

Die gefärbten Petrinetze (*coloured petri net*, CPN) nach Jensen ([Jensen 1990, Jensen 1992, Jensen 1994, Jensen 1997]) wurden vor allem zur angewandten Modellierung entwickelt. Sie stellen eine Weiterentwicklung der Pr/T-Netze dar, die sich aber von der rein prädikatenlogischen Repräsentation entfernt hat. Die Ähnlichkeit zeigt sich vor allem bei der Rolle von Variablen und Variablenbelegungen (*Bindungen*). Im Standard-Werkzeug zur Bearbeitung und Simulation von gefärbten Petrinetzen, DESIGN/CPN ([Design/CPN 2000]), wird die funktionale Sprache Design/ML eingesetzt, um Wächter und die Wirkung von Transitionen zu beschreiben.

Im folgenden soll eine Syntax und Semantik für gefärbte Petrinetze vorgestellt werden, die der von DESIGN/CPN sehr ähnlich ist. Die hier vorzustellende Syntax und Semantik bezieht sich auf das von Kummer und dem Autoren entwickelte Petrinetz-Werkzeug RENEW ([Kummer und Wienberg 2000]), das eine Variante gefärbter Netze als einen Sonderfall unterstützt, in dem die speziellen Erweiterungen für Referenznetze ungenutzt bleiben. Da die Erweiterungen in bezug auf Referenznetze und Feature Structures mit diesem Werkzeug umgesetzt wurden, scheint es naheliegend, auch die grundlegenden gefärbten Netze in der Variante von RENEW einzuführen. Auf wesentliche Unterschiede zu den gefärbten Netzen nach Jensen und DESIGN/CPN wird an gegebener Stelle hingewiesen.

Abbildung 3.18 zeigt ein gefärbtes Petrinetz in RENEW-Syntax, übernommen aus [Kummer und Wienberg 2000]. Das Netz soll dem Leser helfen, die folgenden allgemeinen Ausführen anhand eines Beispiels nachzuvollziehen.

Gefärbte Petrinetze bestehen wie alle höheren Petrinetzen aus Stellen, Transitionen und Kanten, die beschriftet sein können. Die Erweiterungen spezieller Formalismen werden üblicherweise durch spezielle Anschriften ausgedrückt und führen üblicherweise keine neuen graphischen Elemente ein.

Ein- und Ausgangskanten verhalten sich wie aus anderen Petrinetzformalismen

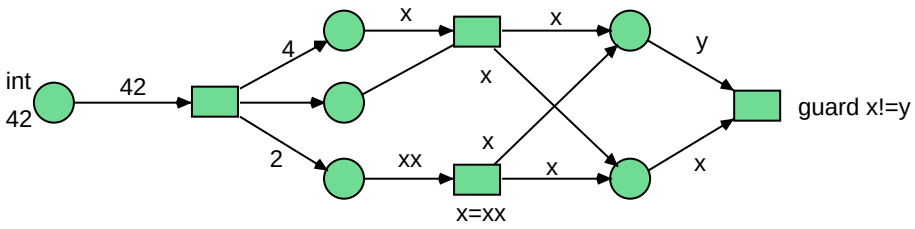

Abbildung 3.18: Ein gefärbtes Petrinetz in Renew-Syntax aus [Kummer und Wienberg 2000], S. 22.

bekannt und konsumieren bzw. produzieren Marken auf den Ein- bzw. Ausgangsstellen. Im Unterschied zu Design/CPN werden Kanten nicht mit Multimengen-Ausdrücken, sondern mit Ausdrücken beschriftet, die einen einzelnen Wert zurückliefern. Um mehrere Markenbindungen an einer Kante zu spezifizieren, müssen mehrere einzelne Anschriften angegeben oder die Ausdrücke mit Semikolons getrennt werden. Damit ist die Anzahl der Marken, die über eine Kante fließen, in Renew konstant. Kanten, bei denen die Anzahl der Marken vom Ergebnis der Ausdrucksauswertung abhängt, werden in Renew als eine spezielle Kantenart (flexible Kanten) unterstützt. Diese Trennung wird bewußt hervorgehoben, da flexible Kanten die Abbildung der Netzstruktur auf ein ungefärbtes Netz (S/T-Netz) und damit die Analyse von kausalen Abhängigkeiten in gefärbten Netzen erschweren.

Weiterhin gibt es in Renew *Reservierungskanten*, die sich wie eine Kombination aus Ein- und Ausgangskante mit derselben Anschrift verhalten. Verschiedene Formalismen führen noch weitere Kantenarten ein. In [Christensen und Damgaard Hansen 1993] werden Test- und Inhibitorkanten vorgeschlagen, die auch in Renew aufgenommen wurden (siehe nächster Abschnitt).

Im folgenden werden die verschiedenen Anschriften vorgestellt, die Netzelemente eines gefärbten Petrinetzes tragen können. Dabei wird nicht über die konkrete Syntax gesprochen, in der solche Anschriften notiert werden.

Zunächst kann jeder Stelle und Transition ein Name zugeordnet werden, der aber keinerlei Einfluß auf die operationale Semantik des Netzes hat.

Stellen kann eine beliebige Anzahl von Ausdrücken zugeordnet werden, die deren Anfangs- oder auch Initialmarkierung bestimmen (in Design/CPN wird die Initialmarkierung wiederum durch eine Multimenge angegeben). Wenn die Simulation des gegebenen Netzes gestartet wird, werden die Ausdrücke ausgewertet und jeder Ausdruck bestimmt den Wert einer Marke, die als Initialmarkierung in der entsprechenden Stelle abgelegt wird. Außerdem erhält jede Stelle einen Typ, der bestimmt, welche Art von Marken in dieser Stelle vorkommen darf. In Formalismen, in denen von einer Typhierarchie ausgegangen wird (beispielsweise in Renew, aber nicht in Design/CPN), dürfen die aktuellen Marken auch einen Subtyp des Stel-

lentyps aufweisen. Wenn der Typ einer Stelle weggelassen wird, ist in Formalismen mit Typhierarchie der allgemeinsten Typ gemeint, der somit jede Art von Marke erlaubt.

Auch Kanten werden mit Ausdrücken beschriftet. Wenn eine Transition schaltet, werden durch die Kanten Marken mit den Werten bewegt, wie sie den ausgewerteten Ausdrücken entsprechen. Eine unbeschriftete Kante konsumiert, erzeugt, reserviert oder testet eine anonyme Marke, das *black token* (in DESIGN/CPN ist die Standardanschrift einer Kante die leere Multimenge, was leicht zu Modellierungsfehlern führt).

Transitionen schließlich können verschiedene Ausdrücke tragen, die anhand ihrer Syntax unterschieden werden. Neben normalen Ausdrücken sind hier Wächter (*guards*) erlaubt, die aus einem Ausdruck mit boolschem Ergebnistyp bestehen. Eine Transition ist nur dann aktiviert, wenn alle Wächter zu *wahr* ausgewertet werden. Für Seiteneffekte von Schaltvorgängen sind spezielle Anschriften vorgesehen, die erst beim Schalten der Transition ausgewertet werden, nicht bereits bei der Überprüfung auf Aktivierung. In RENEW werden solche Anschriften durch das Schlüsselwort `action` eingeleitet, in DESIGN/CPN gibt es zu diesem Zweck *Code-Regionen*.

RENEW unterstützt neben Test- und Reservierungskanten auch Inhibitorkanten. Diese erweiterten Kantenarten sind nicht auf den Einsatz in gefärbten Netzen beschränkt und werden im nächsten Abschnitt genauer beschrieben. Auf andere Erweiterungen von RENEW kommen wir in Abschnitt 3.3.3 zurück.

3.2.4 Erweiterte Kantenarten

Um die Ausdrucksmächtigkeit zu erhöhen, wurden vor allem für höhere Petrinetze zahlreiche Erweiterungen vorgeschlagen. Ein Überblick über verschiedene Kantenarten in gefärbten Petrinetzen gibt [Lakos und Christensen 1993]. Hier sollen kurz die in [Christensen und Damgaard Hansen 1993] eingeführten und im Petrinetzsimulator RENEW (siehe Abschnitt 3.5) verwendeten erweiterten Kantenarten Reservierungs-, Test- und Inhibitorkante vorgestellt werden.

Eine *Reservierungskante* hat dieselbe Wirkung wie eine Eingangs- und eine Ausgangskante mit derselben Anschrift zu derselben Stelle, was diese Stelle zu einer Nebenbedingung der Transition macht. Während diese Abkürzung bei S/T-Netzen nicht viel Ersparnis bringt, kann sie bei gefärbten Netzen mit aufwendigeren Kantenanschriften Modelle stark vereinfachen und übersichtlicher machen. Eine Reservierungskante bindet während des Schaltens einer Transition eine Marke aus der entsprechenden Stelle, was in der Petrinetzliteratur als Nebenbedingung bezeichnet wird (siehe Definition B.2).

Die *Testkante* verhält sich sehr ähnlich, der Unterschied zur Reservierungskante besteht im Verhalten bei nebenläufigem Zugriff auf dieselbe Marke. Während eine Marke stets nur von einer Reservierungskante zur Zeit gebunden werden kann, ist dies für *beliebig viele* Testkanten gleichzeitig möglich. Dadurch läßt sich das in

Abschnitt 3.2.2 vorgestellte Leser-Schreiber-Problem elegant darstellen, indem man schreibende Prozesse durch Reservierungs- und lesende durch Testkanten repräsentiert. Testkanten werden auch dann eingesetzt, wenn eine Marke eine beliebig oft wiederverwendbare Ressource wie zum Beispiel Information repräsentiert.

Solange nur eine Schaltfolgensemantik ohne Synchronisation und Schaltdauer von Transitionen betrachtet wird, ist der Unterschied zwischen Reservierungs- und Testkante minimal, da dann nur das unteilbare Schalten einzelner Transitionen betrachtet wird. Nur, wenn eine Transition mehrere Testkanten zu derselben Stelle besitzt, gibt es dann überhaupt einen semantischen Unterschied. Es werden aber im folgenden Abschnitt erweiterte Semantiken und Formalismen vorgestellt, in denen der Unterschied zwischen Reservierungs- und Testkanten eine wichtige Rolle spielt.

Eine *Inhibitorkante* stellt in gewissem Sinne eine Negation der üblichen Kantensemantik dar. Die Aktivierung einer Transition bei Eingangs-, Reservierungs- und Testkanten hängt von der *Existenz* einer Marke in einer Stelle ab. Dagegen ist eine Transition mit einer Inhibitorkante zu einer Stelle in S/T-Netzen genau dann aktiviert, wenn die Stelle unmarkiert ist. In gefärbten Netzen prüfen die üblichen Kanten die Existenz von Marken, welche die durch Wächter gegebenen Eigenschaften erfüllen. Eine Inhibitorkante in einem gefärbten Netz prüft analog, ob auf der mit der Kante verbundenen Stelle *keine* Marke vorhanden ist, welche die gegebenen Eigenschaften erfüllt.

Ohne Inhibitorkanten sind Petrinetze nicht in der Lage, Bedingungen auszudrücken, welche sich auf *alle* Marken einer Stelle beziehen. Während diese stellenglobale Sicht die Modellierung zum Teil vereinfacht, erweist sie sich für die Theorie als Nachteil: Durch die Erhöhung der Ausdrucksmächtigkeit sind viele interessante Probleme in Petrinetzen mit Inhibitorkanten wie die Frage der Erreichbarkeit einer Markierung nicht mehr entscheidbar ([Christensen und Damgaard Hansen 1993]). Weiterhin zerstören Inhibitorkanten die wichtige Eigenschaft von Petrinetzen, eine genaue Betrachtung der Nebenläufigkeit und Kausalität in Prozessen zu erlauben ([Christensen und Damgaard Hansen 1993]).

3.3 Netze in Netzen

Great fleas have lesser fleas
Upon their backs to bite 'em.
And lesser fleas have lesser still.
And so *ad infinitum*
Jonathan Swift

Die Idee der Netze in Netzen hat ihren Ursprung in den von Valk definierten Auftrags-Verkehrs-Netzen ([Valk 1987, Jessen und Valk 1987]). In höheren Petrinetzen werden komplexe Objekte als Marken in Petrinetzen verwendet, die beispielsweise durch abstrakte Datentypen spezifiziert werden können. Um einen einheitli-

cheren Formalismus zu erhalten und Prozesse auf verschiedenen Ebenen verwenden zu können, entstand mit Auftrags-Verkehrs-Netzen ein Formalismus, bei dem die Marken selbst wiederum *aktive Objekte*, in diesem Fall Auftragssysteme, sind.

Als Verallgemeinerung wurde der Ansatz so weiterentwickelt, daß die aktiven Objekte wiederum durch Petrinetze beschrieben werden. Um einen klaren formalen Ansatz zu erreichen, wird in [Valk 1996] mit einfachen Modellen begonnen, die schrittweise erweitert werden. Valk spricht von einem *Systemnetz*, das *Objektnetze* als Marken enthält. Das Systemnetz kann je nach Komplexität des Modells genau ein Objektnetz in genau einem Zustand, ein Objektnetz in mehreren Kopien oder mehrere unterschiedliche Objektnetze enthalten, wobei der letzte Fall wiederum danach unterteilt wird, ob ein Objektnetz in mehreren Zuständen auftreten kann. System- und Objektnetz können sich durch eine *Interaktionsrelation* an bestimmten Transitionen synchronisieren. In erweiterten Modellen wird auch eine Synchronisation von Objektnetzen untereinander betrachtet. In einem weiteren Modell ([Valk 1996]) wird erlaubt, daß die Marken eines Objektnetzes wiederum für ein weiteres Objektnetz stehen können, wodurch ein n-Ebenen-Modell entsteht. In diesem Zusammenhang wird der Begriff der „Netze in Netzen“ verwendet. Valk spricht für jede Ebene von einem System- oder Umgebungsnetz, wenn dieses andere Netze als Marken enthält und von einem Objekt- oder Markennetz, wenn dieses in einem anderen Netz als Marke enthalten ist. In einem n-Ebenen-Modell kann also insbesondere ein Netz die Rolle des System- und des Markennetzes gleichzeitig einnehmen. Abschnitt 3.3.1 stellt den Ansatz der System-/Objektnetze genauer vor.

Aktive Objekte als Marken in Petrinetzen werden in [Farwer 2000a] ebenfalls betrachtet. Dort werden *Linearlogische Petrinetze* (*linear logic Petri nets*, LLPN) vorgestellt, die linearlogische Formeln als Transitionsanschriften und als Marken verwenden. Linearlogische Formeln können insofern als aktive Objekte betrachtet werden, da sie nach der Schaltregel für LLPN ohne Zutun einer Transition Ableitungsschritte durchführen können. Da die Lineare Logik die Mächtigkeit von S/T-Netzen abdeckt, kann mit diesem Ansatz eine alternative Semantik für die EOS gegeben werden. Linearlogische Petrinetze werden in Abschnitt 3.3.2 vorgestellt.

Inspiriert von diesem und anderen objektorientierten Petrinetz-Formalismen wurden von Kummer die *Referenznetze* entwickelt ([Kummer 1998, Kummer und Wienberg 1999, Kummer 2001]). Hier sind die Marken eines Netzes nicht andere Netze, sondern *Referenzen auf* andere Netze. Das System besteht aus einer Menge von Netzen (genauer: Netzexemplaren), die sich auf einer Ebene befinden. Hierarchien und andere Relationen zwischen Referenznetzen entstehen erst durch die Referenzmarken, die sie enthalten. Abschnitt 3.3.3 enthält eine Einführung in Referenznetze, deren Konzepte in Abschnitt 4.4 als Erweiterungen für den hier vorgeschlagenen Netzformalismus eingesetzt werden.

In den verschiedenen Ansätzen zu Netzen in Netzen tritt ein grundlegendes Phänomen bei der Behandlung von Objekten immer wieder auf: Bei modifizierbaren Objekten und vor allem bei aktiven (sich selbst modifizierenden) Objekten gibt

einen Unterschied zwischen einer Referenz auf ein Objekt und dem Objekt selbst. Damit einher geht der Unterschied zwischen Identität und Gleichheit von Objekten. In verteilten Systemen entspricht dies der Frage, ob verteilte Kopien oder entfernte Referenzen betrachtet werden (siehe Abschnitt 1.1.1). Abschnitt 3.3.4 diskutiert diese beiden Semantiken, die hier als Wert- und Referenzsemantik bezeichnet werden. Die bisherigen Modelle zu Netzen in Netzen haben sich für eine der beiden Semantiken entschieden. In Abschnitt 4.2.5 wird das Thema nochmals aufgegriffen und gezeigt, daß mit den hier vorgestellten elementaren FSNets beide Semantiken innerhalb eines Modells verwendet werden können.

3.3.1 Elementare Objektsysteme

Bei den elementaren Objektsystemen (EOS) handelt es sich um Petrinetze, deren Marken wiederum Netze sind, genauer gesagt elementare Netzsysteme. In [Valk 1998] und [Valk 1999] wird der Stand der Forschung im Bereich der elementaren Objektsysteme detailliert dargestellt, wobei der letztere Beitrag einen Schwerpunkt auf formale Untersuchungen setzt.

In [Valk 1991] werden zunächst elementare Netzsysteme (EN-Systeme, also markierte B/E-Netze) für System- und Markennetz eingesetzt, weshalb diese Netzklasse auch *unäre elementare Objektsysteme* (unäre EOS, *unary elementary object systems*) genannt wird. In [Valk 1996] werden zunächst *unäre EOS* (*unary EOS*) eingeführt, in denen das Systemnetz ebenfalls ein EN-System ist und sich alle Marken auf dasselbe Objektnetz beziehen.

Elementare Objektsysteme bestehen aus einem Systemnetz und einem oder mehreren Objektnetzen. Anschaulich gesprochen enthält das Systemnetz die Objektnetze als Marken. Man kann sich demnach das Systemnetz als höheres Petrinetz vorstellen, das Marken vom Typ „elementares Netzsystem“ enthält. Diese Vorstellung vernachlässigt aber eine wichtige Eigenschaft der elementaren Objektsysteme: Während in höheren Petrinetzen wie gefärbten oder algebraischen Netzen die Marken Datenobjekte sind, die von den Transitionen manipuliert werden, sollen durch die Objektnetze *aktive Objekte* modelliert werden, die auch ohne Zutun des Systemnetzes ihren Zustand ändern können.

Abbildung 3.19 zeigt ein Beispiel für ein elementares Objektsystem aus [Valk 1999] (dort in Abbildung 10 mit anderer Anfangsmarkierung). Links ist das Objektnetz (ON), rechts das Systemnetz (SN) zu sehen. Der Pfeil zeigt an, daß in der abgebildeten Markierung das Objektnetz als Marke auf der Stelle `p1` des Systemnetzes liegt. Durch eine *Interaktionsrelation* werden Paare von Transitionen in System- und Objektnetz bestimmt, die nur synchron schalten dürfen. Graphisch wird diese Relation durch gleich Interaktionsmarkierungen in spitzen Klammern (<i*n*>) angegeben. Es gibt dadurch in elementaren Objektnetzen insgesamt drei verschiedene Arten des Schaltens von Transitionen: Autonomes Schalten des Systemnetzes (*Transport*) oder des Objektnetzes (*objektautonomes Schalten*) oder eine *Interaktion*

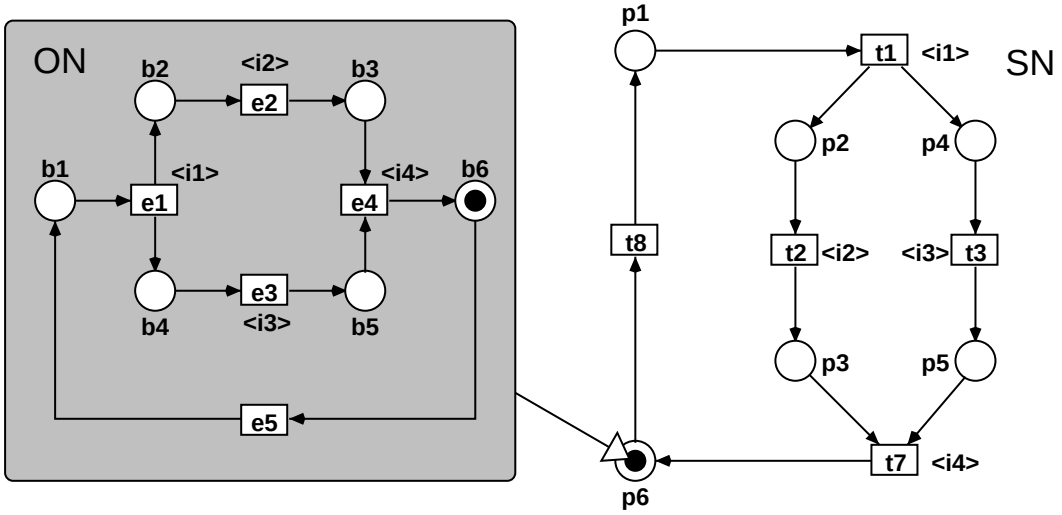

Abbildung 3.19: Ein elementares Objektsystem aus Objektnetz (ON, links) und Systemnetz (SN, rechts) aus [Valk 1999].

zwischen System- und Objektnetz. In [Farwer 1999] (siehe auch nächster Abschnitt) wird weiterhin betrachtet, daß objektautonomes Schalten und Transport gleichzeitig stattfinden.

Elementare Objektsysteme werden in [Valk 1999] in verschiedenen Varianten betrachtet: Zunächst werden unter dem Namen *unäre elementare Objektsysteme* (*unary elementary net systems*) Systeme betrachtet, in denen nur ein einziges Objektnetz existiert, wenn auch verschiedene Marken unterschiedliche Prozesse dieses Objektnetzes repräsentieren können. Eine andere Variante erlaubt verschiedene Objektnetze, dafür wird aber das Systemnetz auf einen Zustandsautomaten beschränkt, so daß Objektnetz-Marken nicht kopiert werden können und somit das Problem der verteilten Kopien nicht betrachtet werden muß.

Das einfachste Modell für elementare Objektsysteme stellt System- und Objektnetze als eigenständige Petrinetze auf einer Ebene nebeneinander und sorgt lediglich für eine Synchronisation der in Interaktionsrelation stehenden Transitionen. Die Markierung eines solchen Netzsystems kann als *Bi-Markierung* notiert werden, die sich als ein Paar aus der Markierung des Systemnetzes und der Markierung des Objektnetzes ergibt. Falls verschiedene Objektnetze verwendet werden, muß für jedes Objektnetz eine individuelle Markierung angegeben und die Markierung des Systemnetzes entsprechend eingefärbt werden.

Die Bi-Markierung reicht aus, solange Objekte nicht kopiert werden bzw. solange Kopien nicht wieder zusammengeführt werden sollen. Stellt man sich dagegen mehrere Marken im Systemnetz als verteilte Kopien *eines* Objektnetzes vor, so kann

aus einer Bi-Markierung nicht immer erkannt werden, ob diese Kopien ihren Zustand *konsistent* verändert haben. Eine Zustandsänderung wird dabei dann als konsistent aufgefaßt, wenn Konflikte in den Prozessen der Kopien gleich gelöst wurden. Valk gibt in [Valk 1998] Beispiele für inkonsistente Zustandsänderungen an Kopien von Objektnetzen, die anhand der Bi-Markierung nicht erkannt werden können.

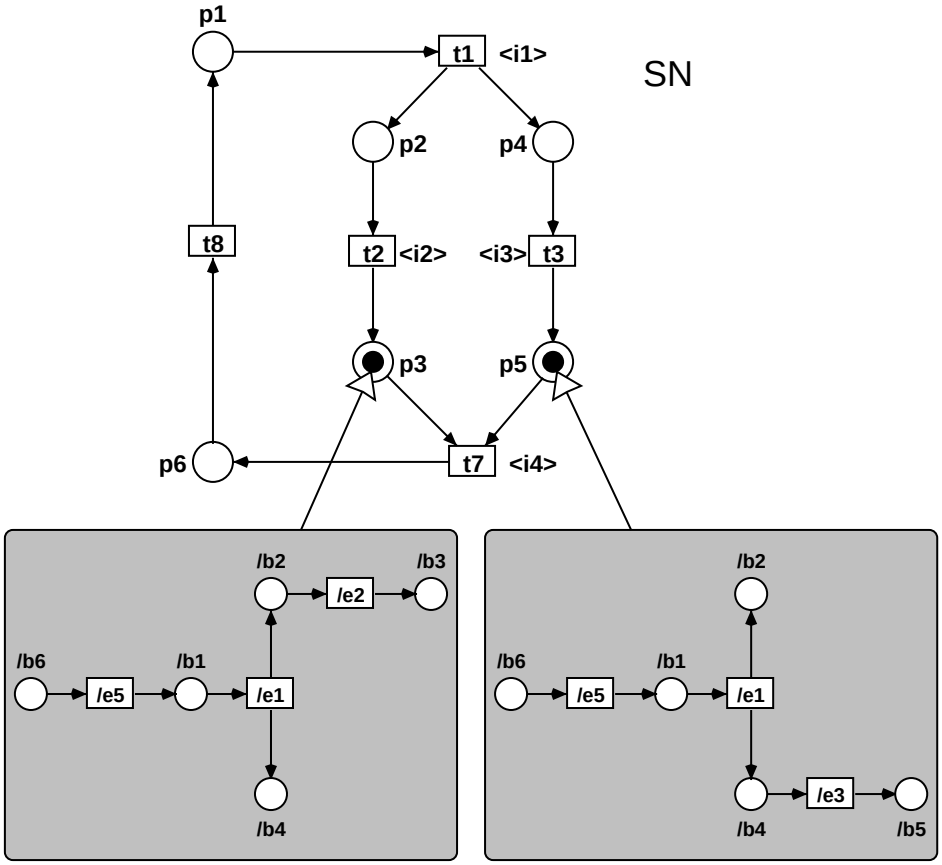

Abbildung 3.20: Eine Prozeßmarkierung des Systemnetzes aus Abbildung 3.19 (aus [Valk 1999]).

Um solche Inkonsistenzen zu erkennen, fehlt der Bi-Markierung eine Historie, auf welche Weise der repräsentierte Zustand erreicht wurde. Diese Historie wird in [Valk 1999] mit dem komplexeren Modell der *p-Markierung* erreicht, indem nicht Markierungen, sondern Prozesse der Objektnetze als Marken im Systemnetz verwendet werden. Abbildung 3.20 zeigt das Systemnetz aus Abbildung 3.19 mit Prozeßmarken des Objektnetzes in den Stellen `p3` und `p5`. In beiden Prozessen kann man erkennen, daß `e5` und `e1` geschaltet haben, der Prozeß in `p3` wurde durch `e2` erweitert, wogegen im Prozeß in `p5` Transition `e3` geschaltet hat. Die Transitionen

und Stellen der Prozesse selbst sind hier unbenannt, so daß sich als jeweiliger Name ein Schrägstrich gefolgt vom Namen des Netzelements aus dem Objektnetz ergibt.

Eine Transition im Systemnetz, die mehrere Marken erzeugt, erstellt Kopien der Prozeßmarken ihres Vorbereichs. Die Kopien können unabhängig voneinander weiterschalten, wodurch der Prozeß des Objektnetzes ergänzt wird. Sollen solche verteilten Kopien nun durch eine Transition des Systemnetzes wieder zusammengeführt werden, so wird die kleinste obere Schranke (*least upper bound*, lub) aller zu vereinigender Prozesse als resultierender Prozeß gewählt. Es handelt sich also um eine *Unifikation von Prozessen*. Da nicht alle Prozesse eine kleinste obere Schranke besitzen, kann diese Unifikation scheitern. Dies ist genau dann der Fall, wenn in den beteiligten Prozessen inkonsistente Entscheidungen getroffen wurden, also Konflikte auf unterschiedliche Weise gelöst wurden.

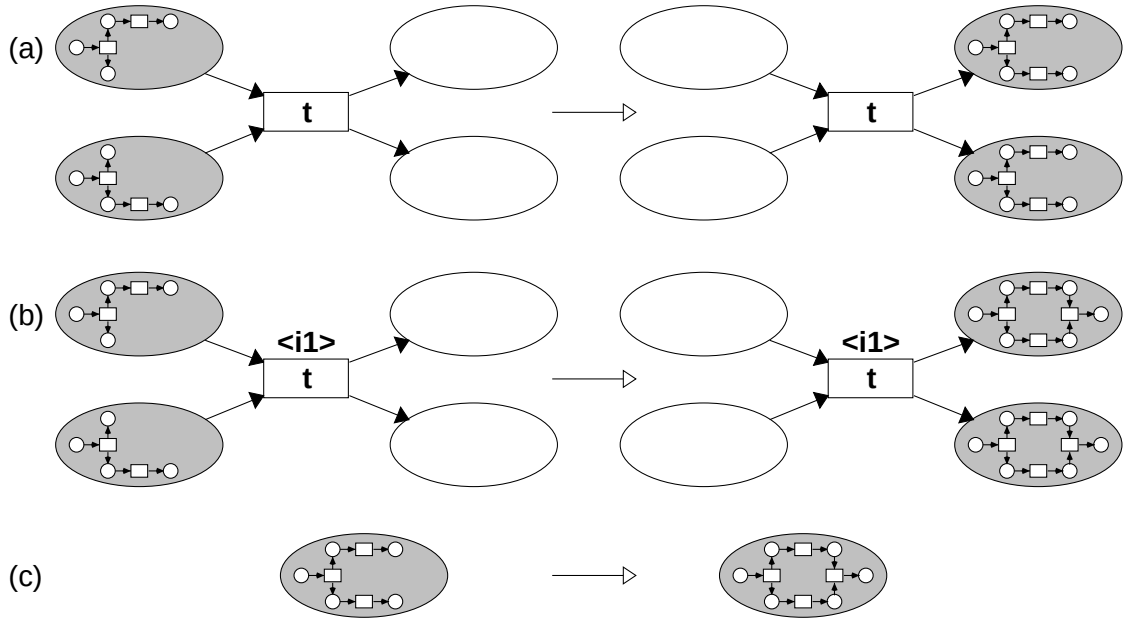

Abbildung 3.21: Beispiele für die Schaltregel für EOS mit Prozeßmarkierungen (aus [Valk 1999], dort Abbildung 14): (a) Unifikation und Kopieren von Prozessen, (b) zusätzliche Interaktion und (c) objektautonomes Schalten.

Die Schaltregel für EOS mit Prozeßmarkierungen wird nun anhand der Beispiele in Abbildung 3.21 ([Valk 1999], dort Abbildung 14) genauer beschrieben. Fall (a) zeigt, wie verschiedene Prozesse im Vorbereich der Transition (links) beim Erzeugen der Folgemarkierung (rechts) unifiziert und kopiert werden. In Fall (b) findet zusätzlich eine Interaktion zwischen System- und Objektnetz statt, so daß das Ergebnis der Unifikation gleichzeitig durch ein weiteres Transitionsvorkommen ergänzt wird. Eine Interaktion ohne Unifikation stellt einen Sonderfall dieses Falls dar und

ist nicht nochmals dargestellt. Ebenfalls nicht dargestellt ist der Sonderfall eines Transports, bei dem *eine* Prozeßmarke nach der normalen Petrinetz-Semantik konsumiert und ohne Veränderung in einer anderen Stelle abgelegt wird. Fall (c) zeigt einen objektautonomen Schaltvorgang (siehe oben).

Die für das Zusammenführen verteilter Kopien nötige Unifikation von Prozessen wird in den Arbeiten von Valk über eine partielle Ordnung auf Prozessen definiert. Damit erhält man eine deklarative, nicht aber eine konstruktive Definition der Unifikation von Prozessen. Da es sich bei Prozessen auch um Graphen handelt, kann in Abschnitt 4.3 die Unifikation von Prozessen auf die in Abschnitt 2.3.3 für Feature Structures eingeführte Graphunifikation zurückgeführt werden. Damit läßt sich der dort beschriebene Unifikationsalgorithmus für die effektive Berechnung der kleinsten oberen Schranke der Prozesse heranziehen und in der Implementierung von FSNets (siehe Abschnitt 4.5) nutzen. Der dort vorgestellte Feature-Structure-Modus von Renew stellt somit einen Prototyp des ersten Simulators für EOS dar.

EOS stellen einen Formalismus mit solider formaler Grundlage dar, der Konzepte der Objektorientierung mit Petrinetzen auf eine neue Weise verbindet. Dabei standen bisher theoretische Ergebnisse im Vordergrund, so daß auf eine Erweiterung des Formalismus um praxisorientierte Eigenschaften für die Modellierung realistischer Problemstellungen weitestgehend verzichtet wurde. Dieses Ziel wird im Bereich der Petrinetze vor allem von den gefärbten Petrinetzen verfolgt, die eine große Anzahl praktischer Anwendungen vorweisen können. Einen guten Überblick dazu bietet [Jensen 1997].

In Abschnitt 4.3 werden EOS mit dem in dieser Arbeit vorgeschlagenen Ansatz der FSNets dargestellt. Dadurch ergibt sich auf natürliche Weise ein n-Ebenen-Modell, in dem die Marken ab einer beliebigen Ebene als Daten, also als gefärbte Marken im Sinne der CPN, aufgefaßt werden können. Dadurch ist eine Modellierung von Netzen in Netzen mit individuellen Marken möglich, die den Formalismus wesentlich interessanter für praktische Anwendungen macht.

Aus der allgemeinen Idee der Netze in Netzen sind zwei weitere Petrinetzformalismen hervorgegangen, die im folgenden betrachtet werden sollen.

3.3.2 Linearlogische Petrinetze (LLPN)

Lineare Logik ([Girard 1987]) ist eine Erweiterung der klassischen Logik, in der nicht nur Aussagen, sondern auch Ressourcen repräsentiert werden können. Bereits ein Fragment der Linearen Logik genügt, um eine rein logische Semantik für Petrinetzen zu definieren ([Martí-Oliet und Meseguer 1989]). Farwer nutzt diese Definition in [Farwer 1998a, Farwer 1998b, Farwer 1999], um eine logische Semantik für Netze in Netzen zu konstruieren. Mit der Darstellung von Petrinetzen als linearlogische Formeln ist es möglich, die Objektnetze eines elementaren Objektsystems (siehe Abschnitt 3.3.1) in linearlogische Formeln zu übersetzen und als Marken im Systemnetz zu verwenden. Diesen Ansatz definiert Farwer als *Linearlogische Petrinetze* (LLPN).

Im folgenden werden kurz die zur Darstellung von Petrinetzen benötigten Konnektoren der Linearen Logik informal eingeführt. Die Abbildung eines S/T-Netzes auf eine linearlogische Formel wird an einem Beispiel verdeutlicht. Sodann werden LLPN intuitiv eingeführt und deren Verbindung zum hier vorgeschlagenen Ansatz der FSNets diskutiert.

Lineare Logik und S/T-Netze

Da eine Einführung in Lineare Logik über die Zielsetzung dieser Arbeit hinausgeht, werden nur kurz die Konnektoren vorgestellt, die für die Darstellung von Petrinetzen mit Linearer Logik benötigt werden. Wir richten uns dabei nach [Farwer 1998b] und [Farwer 1999].

$\otimes$ ***times,tensor*** ist die multiplikative, ressourcensensitive Version der klassischen Konjunktion ($\wedge$). $A \otimes B$ bedeutet, daß die Ressourcen A und B beide gleichzeitig verfügbar sind, während $A \otimes A$ heißt, daß die Ressource A zweimal verfügbar ist. Für das Tensorprodukt $\underbrace{A \otimes \ldots \otimes A}_{n}$ schreiben wir auch A^n.

$\multimap$ ***entails*** ist die multiplikative Implikation. $A \multimap B$ bedeutet, daß eine neue Ressource B entsteht, wenn eine Ressource A verbraucht wird.

! ***of course*** ist ein sogenanntes Exponential, auch Wiederverwendungsoperator genannt, das eine Ressource beliebig häufig verwendbar macht.

$\perp$ ***negation*** wird als hochgestellter unärer Operator hinter einer Formel notiert und beschreibt eine „Schuld“ in Ressourcen. $A^{\perp}$ bedeutet, daß eine Ressource A konsumiert wird.

In [Martí-Oliet und Meseguer 1989] wird eine allgemeine Übersetzungsregel angegeben, wie aus einem markierten S/T-Netz die entsprechende Formel der Linearen Logik konstruiert wird. Das Schalten im Netz wird durch Ableitungsschritte in einem entsprechenden Sequenzenkalkül nachgebildet. Dieser Ansatz wird in [Farwer 1999] auf gefärbte Netze mit erweitert.

Durch die Ressourcenorientierung der Linearen Logik ist diese Übersetzung sehr intuitiv. Marken sind Ressourcen, wobei für jede Stelle des S/T-Netzes ein Ressourcentyp eingeführt wird. Transitionen verbrauchen Ressourcen durch Eingangskanten und erzeugen Ressourcen durch Ausgangskanten. Je nachdem, ob bei einer Transition Ein-, Ausgangskanten oder beide Arten vorhanden sind, wird eine Transition in ein Tensorprodukt von Marken, die Negation eines Tensorprodukts oder eine Implikation übersetzt. Die Implikation stellt den allgemeinsten Fall dar und kann auch ausschließlich verwendet werden, wenn die neutralen Elemente der multiplikativen Konjunktion bzw. Disjunktion $\mathbf{1}$ und $\perp$ verwendet werden. Um das gesamte

markierte S/T-Netz als eine Formeln darzustellen, werden die so entstandenen Teilformeln durch das Tensorprodukt verknüpft. Zu beachten ist hierbei, daß die aus Transitionen entstandenen Teilformeln mit dem *of course*-Operator ! versehen werden müssen, da sonst auch die Transitionen (nicht nur die entsprechenden Marken) während der Anwendung von Regeln des Sequenzenkalküls verbraucht würden.

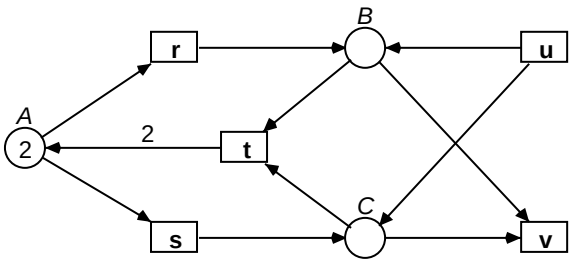

Abbildung 3.22: S/T-Netz, das in eine linearlogische Formel übersetzt werden soll, erweitert aus [Farwer 1999], S. 226.

Die Konstruktion sei anhand des S/T-Netzes in Abbildung 3.22 veranschaulicht, das eine erweiterte Version des in [Farwer 1999] auf S. 226 verwendeten Netzes ist. Wir beginnen mit der Beschreibung der Markierung. Da sich zwei Marken auf der Stelle A befinden, ergibt sich das Tensorprodukt $A \otimes A$ oder auch A^2.

Die Transitionen **r**, **s** und **t** haben Ein- und Ausgangskanten, also ergibt sich für ihr Verhalten eine Implikation, die auf der linken Seite die konsumierten Ressourcen und auf der rechten die produzierten beschreibt. Die Formel, die das Verhalten von **r** beschreibt, ergibt sich zu $A \multimap B$, für **s** zu $A \multimap C$ und für **t** zu $B \otimes C \multimap A^2$.

Die Transition **u** stellt einen Sonderfall dar, da sie nur Ausgangskanten besitzt. Hier reduziert sich die Implikation zu einer Angabe des Tensorprodukts der erzeugten Markierung, in diesem Fall $B \otimes C$. Transition **v** stellt einen analogen Sonderfall für Eingangskanten dar. In diesem Fall wird die Negation verwendet, so daß sich $(B \otimes C)^{\perp}$ ergibt.

Die Formel, die das gesamte markierte Netz repräsentiert, wird wie oben angegeben zusammengesetzt und lautet:

$$A^2 \otimes !(A \multimap B) \otimes !(A \multimap C) \otimes !(B \otimes C \multimap A^2) \otimes !(B \otimes C) \otimes !((B \otimes C)^{\perp})$$

Wenn das Schalten des Netzes wie oben beschrieben durch Ableitungen in einem geeigneten Sequenzenkalkül (siehe [Farwer 2000a, Farwer 2000b]) simuliert wird, ändert sich nur der vordere Teil der Formel, der die aktuelle Markierung beschreibt. Beispielsweise wird dieser durch das Schalten von **r** zur Teilformel $A \otimes B$.

LLPN als logische Semantik für elementare Objektsysteme

Linearlogische Petrinetze gehören zur Familie der höheren Petrinetze. Als individuelle Marken werden hier Formeln eines beliebig festlegbaren Fragments der Linearen Logik verwendet. Wählt man das oben zur Darstellung von S/T-Netzen verwendete Fragment, können die Marken-Formeln (*token formula*) als Petrinetze interpretiert werden und spielen somit eine ähnliche Rolle wie die Objektnetze bei den elementaren Objektsystemen (siehe Abschnitt 3.3.1).

Farwer rekonstruiert in [Farwer 1999] die Semantik der EOS, wobei auch die Interaktion zwischen System- und Objektnetz auf LLPN abgebildet wird. Zunächst wird dies für die Simulation des Schaltverhaltens für die von Valk eingeführten Bi-Markierungen (siehe oben) erreicht, durch zusätzlichen Aufwand aber auch für die Prozeß-Markierungen. Das objektautonome Schalten wird simuliert, indem sich die Marken-Formeln nach dem gewählten Sequenzenkalkül weiterentwickeln dürfen. Als eine Erweiterung gegenüber Valks Ansatz ergibt sich, daß objektautonomes Schalten während eines Transports stattfinden kann.

Weiterhin kann durch LLPN auch die Modifikation der Struktur von Objektnetzen formalisiert werden. Die Netzstruktur ist, wie oben dargestellt, ein Teil der Formel-Marken, der normalerweise nicht verändert wird. Durch die Erweiterung des benutzten Fragments der Linearen Logik kann eine solche Veränderung aber (eingeschränkt) erlaubt werden. Welche inhaltlichen Implikationen dies hat, wird von Farwer nicht weiter untersucht. Er zeigt aber, daß bereits die Erweiterung der Struktur des Objektnetzes von einem elementaren Netzsystem auf ein beliebiges S/T-Netz das Erreichbarkeitsproblem für Objektsysteme unentscheidbar macht.

Der in dieser Arbeit als Modellierungstechnik für verteilte Systeme vorgeschlagene Netzformalismus der FSNets zeigt starken Bezug zu Farwers LLPN. In beiden Ansätzen wird ein höheres Petrinetz definiert, das durch Logik beschriebene Marken enthält. Dies ist im FSNet-Ansatz nicht so deutlich zu erkennen, da Feature Structures hier als Graphen beschrieben werden. [Carpenter 1992] gibt aber eine rein logische Axiomatisierung von Feature Structures.

Eine weitere Gemeinsamkeit der beiden Ansätze besteht darin, daß sie Valks Ansatz der elementaren Objektsysteme durch eine alternative Semantik von einer anderen Seite beleuchten und erweitern. Während Farwer eine Schaltregel auf der Basis eines Sequenzenkalküls nutzt, um Marken aktives Verhalten zuzuordnen, kann mit FSNets das Verhalten von Netzen in Netzen über eine spezielle Prozeßsemantik nachgebildet und in bestimmter Hinsicht erweitert werden, indem Prozesse als Feature Structures dargestellt werden (siehe Abschnitt 4.3). Die Erweiterungen gegenüber Valks EOS liegen in den beiden Ansätzen in unterschiedlichen Bereichen: LLPN erweitern EOS um das Schalten größerer Schritte bei Synchronisationen und erweiterte Netzstrukturen für Objektnetze. Für FSNets wird eine gefärbte, dafür aber auf 1-sichere Netze eingeschränkte Version von Netzen in Netzen definiert. Weiterhin wird die Unterscheidung in Wert- und Referenzsemantik verallgemeinert

sowie durch Stellen-Partitionen in einem Modell gemeinsam zur Verfügung gestellt.

Zumindest in der Zielsetzung läßt sich ein Unterschied der beiden Ansätze festhalten: Während LLPN eine logische Fundierung der Semantik von Netzen in Netzen anstreben, zielen FSNets darauf ab, eine Grundlage für eine in der Praxis einsetzbare Modellierungstechnik bereitzustellen. Durch die Bezüge zu Netzen in Netzen werden in dieser Arbeit zusätzlich zu dieser praktischen Ausrichtung Ergebnisse erzielt, die grundlegende Phänomene bei der Modellierung verteilter Systeme von Seiten der Petrinetztheorie beleuchten.

3.3.3 Referenznetze

Referenznetze sind eine Variante der Netze in Netzen, bei denen Markierungen nicht aus anderen Netzen, sondern aus Referenzen bestehen. Dabei ist zu unterscheiden zwischen dem Basis-Referenznetzformalismus, in dem Marken *ausschließlich* Referenzen auf andere Netzexemplare sind, und dem im Werkzeug Renew verwendeten erweiterten Formalismus, der beliebige Objekte als Marken erlaubt, konkret Java-Objekte.

Eine vollständige Definition und Untersuchung von Referenznetzen wird mit der Dissertation von Kummer ([Kummer 2001]) erscheinen. Bisher wurden grundlegende Definitionen und Beschreibungen von Simulationsalgorithmen in den Arbeiten [Kummer 1998],[Kummer 1999b] und [Kummer et al. 2001] sowie verschiedene Anwendungen im Bereich des Workflow ([Aalst et al. 1999],[Kummer et al. 2000a]), der Agentenmodellierung ([Rölke 1999]) und der Sozionik ([Heitsch et al. 2000]) vorgestellt. Viele der für elementare Objektsysteme durchgeführten Untersuchungen lassen sich auf Referenznetze übertragen, da sich die Semantik der beiden Ansätze für eine bestimmte Klasse von Modellen, bei denen Wert- und Referenzsemantik zusammenfallen, stark ähnelt.

Referenznetze zählen zur Klasse der höheren Petrinetze, da Marken individuell unterscheidbar sind. Die neuen Konzepte, die über gefärbte oder algebraische Petrinetze hinausgehen, sind dynamische Netzexemplare, Netzreferenzen und dynamische Synchronisation von Transitionen über synchrone Kanäle. Da die im Werkzeug Renew hinzugenommenen Erweiterungen gegenüber dem Referenznetzformalism im wesentlichen gefärbten Petrinetzen entsprechen, wird hier auf Abschnitt 3.2.3 verwiesen und nur auf die Referenznetz-spezifischen Eigenschaften eingegangen.

Dynamische Netzexemplare

Das Konzept der Netzexemplare wurde analog zur objektorientierten Programmierung gewählt, wo eine Klasse die Eigenschaften und das Verhalten gleichartiger Objekte spezifiziert. Von einer Klasse werden dynamisch (zur Laufzeit) *Exemplare* (*instances*), auch Ausprägungen genannt, erzeugt. Analog dazu stellen Netze im Referenznetzformalismus lediglich eine Schablone für die zur Laufzeit bestehenden

Netzexemplare (*net instances*) dar. Man spricht bei einer Transition in einem Netzexemplar von einem *Transitionsexemplar* (*transition instance*), analog von Stellen und *Stellenexemplaren* (*place instance*). Nur Transitionsexemplare können schalten und nur Stellenexemplare können Marken enthalten (Stellen enthalten nur Initialmarkierungsausdrücke). Wenn der Unterschied aus dem Kontext eindeutig hervorgeht, werden Transitions- und Stellenexemplare einfach als Transitionen und Stellen bezeichnet.

So wie ein objektorientiertes Programm aus vielen einzelnen Klassen besteht, die individuell instantiiert werden können, besteht ein Referenznetzsystem in der statischen Spezifikation aus einer *Menge von Netzen* mit eindeutigen Namen. Als Ausgangspunkt wird ein Netz bestimmt, von dem gleich zu Beginn der Ausführung ein Exemplar erzeugt wird, *Initialnetz* genannt[6]. Bei der Instantiierung werden sämtliche Initialmarkierungsausdrücke ausgewertet und die entsprechenden Marken in den Stellenexemplaren erzeugt. Im Verlauf der Ausführung können Transitionsexemplare neue Exemplare beliebiger Netze aus der statisch gegebenen Menge erzeugen.

Synchrone Kanäle

Zur Kommunikation zwischen Objekten dienen Nachrichten, realisiert durch Methodenaufrufe. Das entsprechende Konzept bei Referenznetzen stellen die synchronen Kanäle dar. Synchrone Kanäle wurden für Petrinetze erstmals in [Christensen und Damgaard Hansen 1992] vorgeschlagen. Dieser Ansatz wurde für Referenznetze auf Netzexemplare erweitert.

Transitionsexemplare in unterschiedlichen oder auch demselben Netzexemplar können durch einen synchronen Kanal dazu veranlaßt werden, nur synchron (gleichzeitig) zu schalten. Für die Synchronisation ist genau eines der Transitionsexemplare der Initiator. Voraussetzung ist, daß der Initiator Referenzen auf die Netzexemplare kennt, in denen sich die Transitionsexemplare befinden, mit denen eine Synchronisation stattfinden soll. Bei der Synchronisation kann durch Unifikation der Kanalparameter Information ausgetauscht werden. Referenzen auf ein anderes Netzexemplar werden als Markenwerte in Stellen abgelegt und können von einem Netzexemplar auf zwei Arten erlangt werden:

- Das Netzexemplar wird neu erzeugt. Dabei entsteht das Netzexemplar selbst sowie eine Referenz darauf. Dies entspricht dem Aufruf eines Konstruktors in objektorientierten Programmiersprachen.
- Bei einer Synchronisation wird die Referenz über einen Kanalparameter weitergereicht. Dies entspricht den Parametern sowie dem Rückgabewert eines

[6] Viele objektorientierte Sprachen wie C++ und Java lösen das Problem der Initialisierung durch eine statische Methode (`main`), was aber ein prozedurales und damit nicht wirklich objektorientiertes Konzept nötig macht. Referenznetze orientieren sich an Smalltalk, wo eine ausgezeichnete Klasse zu Beginn instantiiert wird.

Methodenaufrufs.

Ein schaltendes Transitionsexemplar kann zusätzlich zu diesen Möglichkeiten eine Referenz auf ein anderes Netzexemplar aus einer Marke erhalten, die bereits in ihrem Netzexemplar vorhanden ist, indem sie diese Marke durch eine Eingangs-, Reservierungs- oder Testkante bindet. Hier werden häufig Testkanten eingesetzt, um eine Netzreferenz mehreren Transitionen gleichzeitig verfügbar zu machen (siehe Abschnitt 3.2.4).

Sämtliche spezielle Konzepte der Referenznetze werden durch Transitionsanschriften ausgedrückt.

Zur Erzeugung neuer Netzexemplare dienen *Instantiierungsanschriften*. Diese bestehen aus einer Variablen, gefolgt von einem Doppelpunkt, dem Schlüsselwort **new** und dem Namen des Netzes, das instantiiert werden soll. Beim Schalten der Transition wird ein neues Exemplar des genannten Netzes in dessen Initialmarkierung erzeugt und eine Referenz darauf der Variablen zugewiesen.

Synchrone Kanäle gibt es in zwei Arten, da wie in der Nachrichtenübermittlung üblich Initiator und Empfänger unterschiedlich behandelt werden. In Referenznetzen wird vom Initiator bestimmt, in welchem Netzexemplar sich die empfangende Transition befinden muß, diese kann hingegen Aufrufe von Transitionen aus beliebigen Netzexemplaren entgegennehmen. Diese Asymmetrie wurde aus folgenden Gründen gewählt:

- Synchrone Kanäle haben den Methodenaufruf objektorientierter Sprachen zum Vorbild. Auch dort gibt es einen Aufrufer und einen Aufgerufenen. Der Aufrufer muß den Aufgerufenen kennen (also über eine Referenz auf diesen verfügen), nicht aber umgekehrt.

- Die Synchronisation über einen Kanal ohne Angabe eines Empfängers wäre im Prinzip möglich, aber nur sehr ineffizient zu realisieren. Eine solche Variante entspräche einem *multicast*, bei dem jeder Interessent sich synchronisieren dürfte, und kann durch An- und Abmelden der Teilnehmer an einem *Ereigniskanal* simuliert werden ([Tu et al. 2000a]).

- Schließlich könnte auch der Empfänger den Absender kennen bzw. einen bestimmten Absender fordern, um die Synchronisation einzugehen. Dies ist aber erfahrungsgemäß oft nicht notwendig und läßt sich bei Bedarf mit der asymmetrischen Lösung leicht nachbilden, indem der Absender eine Referenz auf sich selbst als Paramter im synchronen Kanal übergibt.

Der synchrone Kanal des Initiators wird *Startkanal* (*downlink*), der des Empfängers *Zielkanal* (*uplink*) genannt. Start- und Zielkanal unterscheiden sich syntaktisch nur in der Angabe des Empfängers, die nur im Startkanal enthalten ist. Der eigentliche synchrone Kanal wird durch einen Doppelpunkt angegeben, gefolgt vom

Namen des Kanals und einer in runden Klammern eingeschlossenen Parameterliste, die auch leer sein darf.

Die Schaltregel in Referenznetzen entspricht im Prinzip der in gefärbten Netzen, wobei den synchronen Kanälen eine besondere Bedeutung zukommt. Eine Transition u mit einem Zielkanal $:s_u(p1_u, p2_u, \ldots, pn_u)$ kann niemals von sich aus (*spontan*) schalten. Ebenso kann eine Transition d mit einem Startkanal $x{:}s_d(p1_d, p2_d, \ldots, pn_d)$ nur gemeinsam mit einer Transition mit dem entsprechenden Zielkanal schalten. Genauer müssen folgende Kriterien erfüllt sein:

(a) u muß eine Transition im Netzexemplar x sein,

(b) die Kanalnamen müssen übereinstimmen, also $s_u = s_d$,

(c) d und u müssen *lokal*, also abgesehen von den synchronen Kanälen, nach der Schaltregel für gefärbte Netze aktiviert sein,

(d) die Parameterlisten müssen *unifizierbar* sein, wobei die Unifikation wie bei der Termunifikation in der Prädikatenlogik komponentenweise vorgenommen wird.

Bei der Unifikation der Parameterlisten werden Variablen belegt, indem sie mit Ausdrücken unifiziert werden, deren Wert bereits evaluiert werden kann. Dadurch kann Information über den Kanal übertragen werden, und zwar in beide Richtungen. Während also die Initiierung der Synchronisation asymmetrisch ist, sind beim Informationsaustausch Aufrufer und Aufgerufener vollkommen gleichberechtigt.

Die oben gegebene informale Beschreibung der Schaltregel stellt nur den einfachsten Fall dar, in dem sich genau zwei Transitionen synchronisieren. Die Anzahl der Startkanäle ist in Referenznetzen aber nicht beschränkt, wogegen nur ein Zielkanal pro Transition erlaubt ist. Eine Transition muß sich für jeden ihrer Startkanäle mit einer Transition synchronisieren. Dabei ist genauer zwischen verschiedenen Transitionsvorkommen eines Transitionsexemplars zu unterscheiden. Wie in gefärbten Netzen kann bei entsprechender Aktivierung dasselbe Transitionsexemplar in verschiedenen Modi gleichzeitig schalten. Jeden dieser Schaltvorgänge nennt man wie bei Prozessen (siehe Abschnitt 3.2.2) ein Transitionsvorkommen. Genauer muß es für ein Transitionsvorkommen eines Transitionsexemplars für jeden Startkanal ein Vorkommen eines Transitionsexemplars mit dem entsprechenden Zielkanal geben.

Durch den synchronen Austausch von Information per Unifikation und die hohe Dynamik bei der Auswahl der Synchronisationspartner stellen synchrone Kanäle eine mächtige Erweiterung gegenüber den gefärbten Netzen dar. Das Konzept der synchronen Kanäle wird in die FSNets als Erweiterung aufgenommen (Abschnitt 4.4.1) und in den Modellierungsbeispielen in Kapitel 5 praktisch eingesetzt.

3.3.4 Diskussion: Wert- versus Referenzsemantik

Nachdem verschiedene Ansätze vorgestellt wurden, die das Konzept der Netze in Netzen unterschiedlich interpretieren, sollen diese miteinander in Bezug gesetzt wer-

den. Der wesentliche Unterschied der Ansätze besteht darin, ob die Markennetze sich als Werte in den Stellen des Systemnetzes befinden oder ob die Marken Referenzen auf andere Netze darstellen. Analog zur Terminologie im Bereich der Programmiersprachen wurden dafür die Begriffe *Wert- und Referenzsemantik* geprägt (sieht [Valk 1999], [Valk 2000]).

Bei der Parameterübergabe an Prozeduren in ALGOL-artigen Sprachen kann eine Variable per Referenz (*by reference*) oder als Wert (*by value*) übergeben werden ([Louden 1994]). Im ersten Fall gelten Änderungen, welche die Prozedur an der Belegung der Variablen vornimmt, im gesamten Sichtbarkeitsbereich der Variable „sofort“. Wird der Parameter hingegen als Wert übergeben, wird eine Kopie des zum Aufrufzeitpunkt bestehenden Inhalts der Variable erstellt und nur die Kopie der Prozedur zur Verfügung gestellt. Änderungen an der Kopie haben keinerlei Einfluß auf die Belegung der Variablen in der aufrufenden Prozedur oder überhaupt außerhalb des Sichtbarkeitsbereichs der aufgerufenen Prozedur. Dadurch wird die zusätzliche Komplexität durch mögliche Mehrfachbenennung (*aliasing*) vermieden und es wird eine höhere Unabhängigkeit der Prozeduren erreicht. Andererseits bietet die Übergabe per Referenz den Vorteil, auf Daten ohne den Aufwand des Kopierens zuzugreifen und bietet oft die einzige Möglichkeit, mehrere Ergebnisse einer Prozedur nach außen bekanntzugeben. Einen Kompromiß stellt die Übergabe per Referenz dar, die den modifizierenden Zugriff auf die Variable verbietet (*read-only*, `final`-Parameter in Java).

Gerade in der Objektorientierung wird ausgiebig von der Referenzsemantik Gebrauch gemacht. Durch das Paradigma der Datenkapselung werden die Gefahren der Referenzen zum Teil vermieden: Eine Referenz erlaubt nicht wie in prozeduralen Sprachen den direkten Zugriff auf Daten, sondern nur das Versenden von Nachrichten an den durch die Referenz gegebenen Empfänger. Die direkten Referenzen auf Daten werden zur Unterscheidung auch als Zeiger (*pointer*) bezeichnet. Das Objekt kapselt also seine Daten und steuert den Zugriff auf diese. Trotzdem kann weiterhin der Inhalt eines Objekts von verschiedenen Seiten manipuliert werden, so daß das Alias-Problem nicht beseitigt ist.

Im vorherigen Abschnitt wurden viele der Erweiterungen von Referenznetzen durch Konzepte aus der Objektorientierung motiviert. Deshalb ist es nur natürlich, daß Referenznetze (wie der Name auch schon sagt) eine Referenzsemantik annehmen, wie sie in objektorientierten Programmiersprachen vorherrscht. Dabei wurde Wert darauf gelegt, daß Referenznetze wie ihre objektorientierten Verwandten Daten kapseln. Dies wurde erreicht, indem der Zugriff auf Stellen eines Netzexemplars nur über Transitionen mit synchronen Kanälen möglich ist. Dies entspricht dem Zugriff auf Daten (Exemplarvariablen) über Methoden. Referenzen und synchrone Kanäle erlauben damit eine kompakte Modellierung insbesondere von Problemen, die eine objekt- *und* prozeßorientierte Sichtweise erfordern. Durch den synchronen Zugriff treten viele der Probleme in verteilten Systemen nicht auf bzw. müssen nicht explizit modelliert werden. So stellt es beispielsweise eine starke Abstraktion dar,

einen entfernten Methodenaufruf synchron zu modellieren.

Während einerseits Abstraktion bei der Modellierung erforderlich ist, um sich auf die wesentlichen Aspekte des Modells konzentrieren zu können, so ist es in verteilten Systemen oft gerade die Synchronisation, die im Detail modelliert werden soll. In diesem Fall verbirgt die Referenzsemantik die Komplexität zu stark. Valk zeigt in [Valk 1996], daß ein synchrones Schalten von System- und Markennetz nicht immer der Intuition in verteilten Systemen entspricht. Intuitiv ist Synchronisation nur lokal möglich, nicht über astronomische Distanzen. Hier wird also eine explizite Modellierung mit Wertsemantik nötig, in der es verteilte Kopien von Daten gibt, die sich (asynchron) ändern können. Lokale Änderungen haben in einem verteilten System zunächst nur lokale Wirkung.

Mit der Einführung von verteilten Kopien entsteht sofort das Problem der Konsistenz in verteilten Systemen. Kopien, die dasselbe Objekt an verschiedenen Orten repräsentieren, sollten nicht inkonsistent verändert werden können. In einer Referenzsemantik wird die inkonsistente Änderung von vornherein ausgeschlossen, da immer auf den (zentralen) Originaldaten gearbeitet wird. Dies ist aber in verteilten Systemen mit erheblichen Nachrichtenlaufzeiten nicht realistisch. Akzeptiert man also, daß Kopien auf unterschiedliche Weise verändert werden können, so benötigt man einen Mechanismus, der Konsistenz überprüft und veränderte Kopien wieder zusammenführt.

Valk nutzt zur Lösung des Problems in seiner Wertsemantik der elementaren Objektsysteme Unifikation, in diesem Fall Unifikation von Prozessen. Durch Unifikation wird sowohl Information auf Konsistenz geprüft als auch bei Erfolg als Ergebnis ein Informationsobjekt geliefert, das die Summe der Einzelinformationen enthält. Valk wendet Unifikation auf Prozesse an, um nicht nur zu erkennen, ob der Zustand der Markennetze kompatibel ist, sondern auch, ob die Kopien auf dieselbe Art und Weise in den momentanen Zustand versetzt wurden. Da nur der Prozeß diese „historische“ Information enthält, reicht die im ersten Ansatz vorgeschlagene Bi-Markierung nicht aus.

Interessant ist, daß auch in der Referenzsemantik Unifikation genutzt wird, und zwar beim Informationsabgleich zwischen den Parametern eines synchronen Kanals. Dies entspricht einer synchronen, transaktionalen Einigung über Werte von Variablen, wogegen in der Wertsemantik Unifikation Veränderungen zusammenfaßt, die über einen Zeitraum entstanden sind.

Es scheint, daß Unifikation in verteilten Systemen ein grundlegende Technik darstellt, um die Einigung auf Parameter oder die Zusammenführung von Information zu ermöglichen. Diese Erkenntnis beeinflußte die Entwurfsentscheidungen bei der Definition der FSNets grundlegend. Mit Feature Structures wurde eine Informationsrepräsentation gewählt, die sich besonders zur Unifikation eignet. In Abschnitt 4.2.1 wird gezeigt, welche neuen Erkenntnisse sich dadurch in bezug auf Wert- und Referenzsemantiken gewinnen lassen.

3.4 Geschäftsprozeßmodellierung

Geschäftsprozesse sind im Rahmen einer geschäftlichen Organisation vorkommende Abläufe oder Vorgänge ([Jablonski et al. 1997]). Die Modellierung von Geschäftsprozessen ist weit mehr als eine reine Prozeßmodellierung: Im Firmenalltag spielen Ressourcen (menschliche sowie technische), Daten, Aufgaben und Operationen eine Rolle, die als verschiedene Ebenen in die Geschäftsprozeßmodellierung eingehen ([Aalst et al. 1999]). Allerdings stehen weiterhin die Abläufe oder Prozesse im Mittelpunkt der Betrachtung.

3.4.1 Einführung

Geschäftsprozesse werden aus verschiedenen Gründen modelliert. Zum einen haben sich die Abläufe in einer Firma oft evolutionär entwickelt, so daß sie nur in den Köpfen der Mitarbeiter vorhanden, also de facto undokumentiert sind. Dadurch ist die Einarbeitung neuer Mitarbeiter mühsam und die Ursachen für Fehlverhalten des Systems oder Ansatzpunkte für alternative Vorgehensweisen sind schwierig auszumachen. Somit dient die Geschäftsprozeßmodellierung der Dokumentation der bestehenden Abläufe.

Weiterhin sollen Geschäftsprozesse, wenn sie erst einmal modelliert und damit besser verstanden sind, optimiert werden. Die Optimierung von Geschäftsprozessen ist ein weites Betätigungsfeld für Unternehmensberater, die Firmen damit eine hohe Produktivitätssteigerung versprechen. Um Geschäftsprozesse optimieren zu können, ist ein fundiertes Verständnis der verwendeten Modellierungstechnik eine wichtige Voraussetzung. Hier erweist es sich als vorteilhaft, wenn Geschäftsprozeßmodelle in Form einer *Simulation* ausgeführt werden können, um eine automatisierte Validation und Analyse der Modelle zu ermöglichen.

Nicht zuletzt werden Geschäftsprozeßmodelle auch deswegen erstellt (und optimiert), um den realen Geschäftsprozeß automatisch unterstützen zu können. Erst hier kommt der Begriff *Workflow* ins Spiel, der aber auch oft synonym für „Geschäftsprozeß“ verwendet wird. Durch ausführbare Modelle kann erreicht werden, daß Workflow und Geschäftsprozeßmodell verschiedene Sichten auf ein gemeinsames Modell sind.

Ein Workflow ist nach Definition der Workflow Management Coalition (WFMC, [WFMC 2000]) ein formalisierter Geschäftsprozeß, der durch ein sogenanntes Workflow Management System (WMF-System) automatisch ausgeführt werden kann ([WFMC 1998]). Genauer spricht man von einer *Workflow-Definition* (*workflow definition*), die eine statische Beschreibung eines automatisch auszuführenden Geschäftsprozesses darstellt. Die WFMC-Terminologie nennt weiterhin die konkrete Ausprägung einer Workflow-Definition einen *Fall* (*case*). In [Jablonski et al. 1997] wird von einer Workflow-Instanz, einer Workflow-Ausprägung oder einem Workflow-Exemplar gesprochen. In dieser Arbeit soll der Begriff *Workflow-Exemplar* bevorzugt

werden, da „Fall“ in vielen Kontexten mehrdeutig, das deutsche Wort „Instanz“ in der Objektorientierung als schlechte Übersetzung des englischen Begriffs *instance* gilt und „Ausprägung“ eher ungebräuchlich ist. Weiterhin paßt diese Terminologie am besten zu dem für Referenznetze gewählten Begriff des *Netzexemplars* (siehe Abschnitt 3.3.3).

Eine Aufgabe (*task*) ist ein einzelner Schritt in einer Workflow-Definition, der in einem konkreten Fall zu einer *Arbeitsaufgabe* (*work item*) wird. Ein *Workflow-Teilnehmer* (*workflow participant*) kann aus seinem *Arbeitskorb* (*work basket*) verschiedene Arbeitsaufgaben zur Bearbeitung auswählen. Hier kommen weitere Ressourcen, Daten und Operationen hinzu, die einem Workflow-Exemplar zugeordnet sein können oder zur Bearbeitung benötigt werden. Workflow-Management-Systeme werden dafür oft mit sogenannten Content-Management-Systemen kombiniert, welche die zu bearbeitenden Dokumente verwalten ([Jablonski et al. 1997]). Man spricht hier vom *Datenfluß* (*data flow*) im WFM-System. Daten wie z.B. Attribute von Dokumenten werden unterteilt in solche, die den Ablauf des Geschäftsprozesses beeinflussen (Steuerinformation, *workflow process control data*) und andere, die nur innerhalb externer Applikationen von Bedeutung sind (Applikationsdaten, *application data*). Zumindest Steuerinformation sollte im Workflow modelliert werden können, in einem integrierten System auch Applikationsdaten. Dies ist dann nötig, wenn die Semantik der Operationen, die Aufgaben realisieren, im Modell spezifiziert werden soll.

Workflow-Teilnehmer werden einer oder mehreren Rollen zugeordnet, die bestimmen, welche Aufgaben sie vom WFM-System zugeordnet bekommen oder auswählen dürfen. Die Rollen stellen die verschiedenen Positionen im *Organisationsmodell* des Unternehmens dar und werden oft in Generalisierungsbeziehungen oder Assoziationen miteinander gestellt. Auch eine Modellierung der Attribute von Rollen, Workflow-Teilnehmern, Dokumenten und Ressourcen, die für das Workflow-Management relevant sind, führt zu einem aussagekräftigeren und potentiell besser automatisierbarem Modell.

Zur Spezifikation von Workflow-Definitionen existieren textuelle und graphische Repräsentationen. [Jablonski et al. 1997] gibt einen aktuellen Überblick über verschiedene Techniken und klassifiziert diese anhand von Meta-Modellen nach verschiedenen Gesichtspunkten. Da es für die meisten textuellen Modellierungstechniken eine entsprechende graphische Technik gibt und die Konzepte in beiden Fällen dieselben sind, beschränken wir uns in den konkreten Beispielen auf graphische Techniken.

Alle dem Autoren bekannten graphischen Modellierungstechniken für Geschäftsprozesse verwenden als Basis-Struktur einen *Graphen* aus *Knoten* und *Kanten*.

Eine wichtige Entwurfsentscheidung für eine Workflow-Modellierungstechnik ist die Methode zur Verfeinerung und Wiederverwendung von Workflows. Die meisten Ansätze wie Cosa [COSA 2000], FlowMark [IBM 1996] und die *workflow process*

definition language (WPDL) der WFMC [WFMC 2000] geben eine feste Anzahl von Verfeinerungsebenen vor, es gibt aber auch dynamischere Ansätze wie MOBILE [Jablonski und Bußler 1996]. Bestimmte Knotenarten zeigen an, daß ein Knoten durch einen weiteren Graphen verfeinert wird. Flexibler ist eine Lösung, in der jeder Workflow eigenständig genutzt oder von anderen Workflows instantiiert werden kann. Dadurch ist eine beliebig tiefe Schachtelung und eine flexible Wiederverwendung gewährleistet. Als Nachteil dieser Lösung ist zu nennen, daß im WFM-System eine größere Anzahl von Workflow-Exemplaren existiert, der inhaltliche Zusammenhang von Aufgaben verloren gehen kann und die Modellierer von der Flexibilität dieses Ansatzes zum Teil überfordert sind ([Jablonski et al. 1997]).

Ein weiterer grundsätzlicher Unterschied, der anhand der Beispieltechniken aus Abschnitt 3.1 besonders deutlich wird, ist, ob Zustände *und* Zustandsübergänge des Systems als Knoten des Workflow-Graphen modelliert werden. Dieser Aspekt geht zurück auf die in [Aalst 1998] vorgestellte grundlegende Unterscheidung zwischen zustands- und ereignisorientierter Modellierung. Modellierungstechniken, die nur eine Knotenart kennen, sollen hier *aktivitätenorientierte Modellierungstechniken* genannt werden, da diese Knoten meist als *Aktivitäten* bezeichnet werden. Aktivitätenorientierte Modellierungstechniken erlauben eine kompaktere Darstellung von Prozessen, die sich positiv auf die Übersichtlichkeit vor allem der graphischen Notation auswirkt. Werden Zustände und Zustandsübergänge als separate Knoten modelliert, wie z.B. bei Petrinetzen und EPK, entstehen im Extremfall doppelt soviele Knoten im Workflow-Graphen und in vielen Fällen inhaltliche Redundanzen („Kontonummer überprüfen – Kontonummer ist überprüft“).

Während aktivitätenorientierte Modellierungstechniken in der Kompaktheit der Notation Vorteile zeigen, soll im folgenden begründet werden, daß die Unterscheidung von Zuständen und Aktivitäten zu einer saubereren Semantik und zu einer besseren Darstellung während des Ablaufs von Workflows führt.

Im Aktivitätenmodell wird davon ausgegangen, daß bei Beendigung einer Aktivität sofort eine oder mehrere der Folgeaktivitäten angestoßen wird. Dies ist aber dann nicht der Fall, wenn die Folgeaktivität mehrere Vorbedingungen besitzt, die alle erfüllt sein müssen, also eine Synchronisation durchführt. Dann kann sich das System wie oben beschrieben in einem Zustand befinden, in dem die Vorgänger-Aktivität beendet, die Nachfolger-Aktivität aber noch nicht gestartet ist. Hier sind wiederum zwei Fälle zu unterscheiden: interne und externe Bedingungen.

Als *interne Bedingungen* werden hier kausale Abhängigkeiten zwischen im Prozeßmodell spezifizierten Aktivitäten, also der Steuerfluß bezeichnet. Dies ist der normalerweise bei Synchronisationen betrachtete Fall. Seltener betrachtet werden in Workflows dagegen *externe Bedingungen*, bei denen es um die Verfügbarkeit von Ressourcen geht, die zum Starten der Aktivität nötig sind. Solche Ressourcen sind insbesondere Workflow-Teilnehmer mit den entsprechenden Rollen, welche für die Durchführung der Aktivität verantwortlich sind, aber auch technische Ressourcen wie beispielsweise ein Drucker. Ist eine der geforderten Ressourcen nicht verfügbar,

befindet sich das System ebenfalls in einem wie oben beschriebenen „Zwischenzustand“. Es ist demnach nicht richtig, daß in einem Geschäftsprozeß nur die Aktivitäten zeitbehaftet sind, die (Synchronisations-)Zustände aber zeitlos.

Ein weiteres, oben bereits angedeutetes Problem der aktivitätenorientierten Modellierung ist die mangelnde konzeptuelle Unterscheidung von Nebenläufigkeit und Auswahl. Dies betrifft sowohl die Lesbarkeit der Modelle als auch ihre Semantik. Beim Betrachten eines Modells wie in Abbildung 3.6 ist auf den ersten Blick nicht zu erkennen, welche der in der Aktivität Kredit gewähren zusammenlaufenden Kanten ein Signal geben müssen, damit die Aktivität gestartet werden kann. Es hat sich gezeigt, daß die Zusammenführung nebenläufiger und alternativer Zweige eine der häufigsten Fehlerquellen bei der Modellierung von Prozessen ist ([Aalst 1998]). Die in Abschnitt 3.1.3 beschriebenen Aktivitätsdiagramme von UML zählen im Prinzip zu den aktivitätenorientierten Modellierungstechniken, erlauben aber die explizite Unterscheidung von Nebenläufigkeit und Auswahl durch entsprechende Hilfsknoten (Entscheidungsknoten und Synchronisationsbalken).

Die genannten semantischen Probleme der aktivitätenorientierten Modellierung manifestieren sich auch bei der graphischen Anzeige eines in Ausführung befindlichen Workflows, die nach [Jablonski et al. 1997] zur Simulation von Geschäftsprozessen und zum *Arbeitsablauf-Monitoring* laufender Workflows eingesetzt wird. Es reicht aufgrund der Existenz von „Zwischenzuständen“ nicht aus, anzuzeigen, welche Aktivitäten gerade laufen oder bereits abgearbeitet sind. FlowMark (siehe Abschnitt 3.1.4) löst das Problem durch zusätzliche Anzeige von kleinen Kreisen an den Steuerkonnektoren, wenn diese die Beendigung der Vorgängeraktivität signalisieren. Trotzdem ist ohne einen Blick auf die Herkunft des Steuerflusses nicht zu entscheiden, durch welche Kombination von Signalen über die Steuerkonnektoren eine Aktivität ausführbar ist.

Um Workflow-Modellierungstechniken eine wohldefinierte Semantik zu geben, ist somit eine genaue Bestimmung der kausalen Abhängigkeiten zwischen Aktivitäten nötig. Vor allem können in Modellen, in denen von Zuständen außerhalb der Aktivitäten abstrahiert wird, Nebenläufigkeit und Alternativen nicht sauber unterschieden werden ([Aalst 1998]). Ein Formalismus, der auf die Darstellung von Kausalitäten und Nebenläufigkeit spezialisiert ist, stellen die in den folgenden Abschnitten behandelten Petrinetze dar. Diese bieten sich deswegen als semantische Basis, wenn nicht dank ihrer graphischen Repräsentation sogar direkt als Modellierungstechnik für Prozesse an.

Für die Modellierung von Geschäftsprozessen bzw. Workflows wurden bereits unzählige konkrete Techniken vorgeschlagen. Viele der heute verfügbaren WFM-Systeme verfügen über ihre eigene, proprietäre Modellierungssprache. Als (relativ) werkzeugunabhängige Techniken stehen zur Workflow-Modellierung vor allem die in Abschnitt 3.1 vorgestellten EPK und Aktivitätenmodelle sowie verschiedene Varianten von Petrinetzen zur Verfügung, auf die ab dem nächsten Abschnitt näher eingegangen wird. Auch UML-Aktivitätsdiagramme können zur Geschäftsprozeßmo-

dellierung eingesetzt werden ([MID 2000]). Diese Möglichkeit wird in Abschnitt 5.2.2 in einem Beispiel zur Modellierung ausführbarer Verträge eingesetzt.

3.4.2 Höhere Petrinetze

In diesem Abschnitt werden die Bezüge zwischen Petrinetzen und konkreten Anwendungen in der Prozeßmodellierung beleuchtet. Dafür werden verschiedene Petrinetzmodelle vorgestellt, die auf die Modellierung von Geschäftsprozessen spezialisiert sind oder als allgemein verfügbare Technik in diesem Bereich eingesetzt wurden.

Petrinetze sind von allen Workflow-Modellierungstechniken diejenigen mit der am besten formal untersuchten operationalen Semantik. Für viele der anderen Modelle wurde eine Abbildung auf Petrinetze spezifiziert, um ihnen selbst eine formale operationale Semantik zu geben. Aufgrund der Darstellung von Ereignissen *und* Zuständen können sowohl zustands- als auch ereignisorientierte Modellierungstechniken auf Petrinetze abgebildet werden. Daraus ergibt sich der weitere Vorteil, daß für diese Techniken die theoretischen Erkenntnisse aus jahrzehntelanger Grundlagenforschung im Bereich der Petrinetze zur Verfügung stehen.

Ein möglicher Grund, warum Petrinetze selten direkt als Modellierungstechnik für Workflows eingesetzt werden, ist ihre starke Abstraktion. Petrinetze können beliebige Prozesse modellieren und sind damit nicht auf die spezielle Anwendung „Geschäftsprozesse“ optimiert ([Uthmann 1997, Uthmann und Becker 1998]). Für häufig auftretende Phänomene wie Zeitüberschreitungen (*time-outs*), externe Ereignisse, Ausnahmen, Benutzerinteraktion etc. gibt es bei Petrinetzen keine allgemein anerkannte Notation. [Graaf und Aalst 1998] schlägt entsprechende Erweiterungen in der Petrinetz-Notation vor. Andere Modellierungstechniken wie Zustandsdiagramme (siehe Abschnitt 3.1.2) bieten aber noch bessere abkürzende Notationen, die es erlauben, einen Sachverhalt im Vergleich zu einem Petrinetz mit einer kleineren Anzahl graphischer Elemente auszudrücken.

Aus diesen Gründen findet der Einsatz von Petrinetzen in der Workflow-Modellierung oft „hinter den Kulissen“ statt. Beispielsweise kann ein Workflow in einem Werkzeug als Aktivitätsdiagramm modelliert, intern aber auf ein Petrinetz abgebildet werden, das dann vom Werkzeug zur Analyse und zur Ausführung herangezogen wird (siehe beispielsweise [Verbeek et al. 1999, COSA 2000]). Der Benutzer des Werkzeugs bekommt im Extremfall niemals ein Petrinetz zu Gesicht.

Verschiedene der im vorherigen Abschnitt vorgestellten Erweiterungen werden aber auch direkt für die Modellierung von Geschäftsprozessen eingesetzt.

In [Valk 1996] werden elementare Objektsysteme zur Modellierung von Prozessen mit mehreren Modellierungsebenen eingesetzt. Als Beispiele dienen in [Valk 1998] die flexible Fertigung (*flexible manufacturing*) und die Geschäftsprozeßmodellierung. Das Systemnetz beschreibt die allgemeinen Abläufe, die zwischen verschiedenen Funktionseinheiten stattfinden können. Konkrete Produktionsabläufe bzw. Geschäftsprozesse werden als Objektnetze modelliert. Dies hat den Vorteil, daß die

generelle Zusammenarbeit der Funktionseinheiten getrennt von den konkreten Prozessen modelliert werden, aber beide Ebenen mit derselben Technik beschrieben werden können. Für ein modelliertes Systemnetz aus Funktionseinheiten wird es im allgemeinen viele als Objektnetze modellierte Produktionsabläufe geben. Auch ist zu erwarten, daß sich die Kooperationsstruktur der Funktionseinheiten seltener ändert als die konkreten Produktionsabläufe.

Auch Referenznetze wurden bereits in der Geschäftsprozeßmodellierung eingesetzt. In [Aalst et al. 1999] werden nicht spezielle Workflows modelliert, sondern die generelle Architektur einer Workflow-Management-Systems wird mit Referenznetzen spezifiziert. Weiterhin wurde für das im nächsten Abschnitt vorgestellte Referenznetz-Werkzeug RENEW eine Erweiterung erstellt, welche durch bestimmte Anschriften gekennzeichnete Transitionen als Aufgaben eines Workflows interpretiert und über ein entsprechendes Protokoll die Anbindung einer Workflow-Client-Anwendung an die Ausführungsmaschine von RENEW erlaubt (siehe Abschnitt 3.5.2). In [Müller-Wilken et al. 2000] wird diese Workflow-Erweiterung für RENEW in Kombination mit dem Zugang zu Workflow-Management-Systemen über mobile Geräte eingesetzt. Eine weitere Anwendung im Bereich der Ausführung elektronischer Verträge wird in Abschnitt 5.2.2 genauer behandelt.

Im Projekt DynamicS ([Tu et al. 2000b]) wurde eine Architektur für verhandelnde Agenten auf der Basis flexibler Protokolle entworfen. Zur Modellierung der Protokolle wird eine spezielle Art von Petrinetzen, OO-PAMELA-Netze, eingesetzt. In [Grossler 2000] wurden OO-PAMELA-Netze auf Referenznetze abgebildet und in einer Implementierung gezeigt, daß damit die Agentenprotokolle ausgeführt werden können.

In den folgenden Abschnitten werden mehrere petrinetzbasierte Ansätze zur Geschäftsprozeßmodellierung näher vorgestellt.

3.4.3 Workflow-Netze

Van der Aalst beschreibt in [Aalst 1997], welche Eigenschaften ein Petrinetz erfüllen muß, um als *wohlgeformtes Workflow-Netz* zu gelten. Im wesentlichen orientiert er sich dabei an wohlbekannten Petrinetz-Eigenschaften wie Sicherheit, Verklemmungsfreiheit und *free-choice* (siehe auch [Baumgarten 1990]). Die Untersuchungen haben das Software-Werkzeug WOFLAN hervorgebracht, das nicht-wohlgeformte Workflowstrukturen automatisch erkennt und dabei die Fehlerquelle eingrenzen kann ([Aalst 1999a, Verbeek und Aalst 2000]).

In weiteren Untersuchungen (u.a. [Aalst und Basten 1997, Aalst und Anyanwu 1999]) widmet sich van der Aalst der *Verhaltensvererbung* zwischen Workflow-Netzen und zeigt, daß sich auch hier formale, praktisch anwendbare Ergebnisse erzielen lassen. Verhaltensvererbung zwischen Workflows spielt beispielsweise eine Rolle, wenn ein Workflow in verschiedenen Unternehmen unterschiedlich realisiert werden kann, in jedem Fall aber eine Spezialisierung eines

vereinbarten „abstrakten“ Workflows sein muß. Dies entspricht beispielsweise der Problemstellung des Projekts Crossflow ([Grefen et al. 2000]). Eine weiteres Anwendungsgebiet für Verhaltensvererbung ist die Änderung einer Workflowstruktur *zur Laufzeit*, also während von diesem Workflow Exemplare existieren.

Neben einer semantischen Charakterisierung von spezialisiertem Verhalten wird in [Aalst und Anyanwu 1999] eine Menge von Umformungsregeln definiert, die auf struktureller Ebene einen Erhalt eines minimalen gemeinsamen Verhaltens garantieren.

3.4.4 Prolog-Netze: Deklarative Geschäftsregeln

Eine weitere Variante höherer Petrinetze, die speziell für den Einsatz im Bereich der Geschäftsprozeßmodellierung und -simulation geeignet sind, stellen die im Werkzeug Income ([Promatis 2000, Oberweis et al. 1994]) verwendeten Prolog-Netze dar. Diese Art höherer Petrinetze geht zurück auf die von Oberweis definierten Petrinetze für Datenbankanwendungen ([Oberweis et al. 1987, Oberweis et al. 1986]).

Die Grundidee des Werkzeugs Income ist, Geschäftsprozesse als Petrinetze zu modellieren, deren Transitionen sogenannte Geschäftsregeln enthalten. Im Unterschied zu Design/CPN und Renew (siehe Abschnitt 3.2.3) wird durch die Darstellung als Regel die Verarbeitung der Marken durch eine Transition *deklarativ* gegeben, also ausgedrückt durch die Bedingungen, die zwischen Markierung und Folgemarkierung gelten sollen, nicht durch die expliziten Verarbeitungsschritte, *wie* man zu der Folgemarkierung gelangt. Diese Eigenschaft haben Prolog-Netze mit FSNets gemeinsam, es gibt aber einige Unterschiede im Detail.

Prolog-Netze verwenden Prolog-Terme als Marken und Prolog-Regeln als Transitionsregeln. Die Kanten werden mit Variablen beschriftet, die in den Transitionsregeln vorkommen dürfen.

In Abschnitt 2.3.1 wurde bereits ein grober Vergleich zwischen Feature Structures und Prädikatenlogik diskutiert. Prolog-Terme entsprechen Grundtermen (*ground terms*) der Prädikatenlogik, während Prolog-Regeln das Gegenstück zu Hornklauseln darstellen. Carpenter zeigt im Anwendungsteil von [Carpenter 1992], wie Logikprogrammierung durch Feature Structures simuliert werden kann. Prädikatenlogik läßt sich als ein Spezialfall der Feature Structures auffassen, in dem Atome und Funktoren durch Typen dargestellt werden, zwischen denen keine Vererbungsbeziehungen bestehen. Hornklauseln können durch einen weiteren Typ dargestellt werden (siehe [Carpenter 1992]). Der Ansatz, deklarative Regeln in Transitionen und rekursive Datenstrukturen als Marken einzusetzen, weist also eine starke Ähnlichkeit mit FSNets auf und wird deshalb im folgenden genauer vorgestellt.

Definition von Prolog-Netzen

Wir stellen eine Idealisierung der Prolog-Netze vor, wobei praktische Details der Realisierung im Werkzeug Income vernachlässigt werden.

Ein Prolog-Netz ist ein höheres Petrinetz, bei dem die Transitionen mit Prolog-Regeln, die Kanten mit Variablen und die Stellen mit Term-Mustern beschriftet werden. Term-Muster sind verschachtelte Terme mit Platzhaltern für die Argumente und haben die Funktion einer Typdefinition für die Stelle.

Die Marken werden durch vollständig instanziierte Prolog-Terme (Fakten) dargestellt. In einer Stelle dürfen nur solche Prolog-Terme liegen, die dem der Stelle zugeordneten Term-Muster entsprechen.

Die Menge der Variablen einer Transition bestimmt sich aus allen Variablen, die in der Anschrift einer Kante auftreten, die mit dieser Transition verbunden ist, sowie allen Variablen, die in der Transitionsregel selbst vorkommen.

Eine Transition ist aktiviert, wenn es eine Belegung aller Variablen der Transition gibt, welche die Prolog-Regel der Transition erfüllt und unter der die Eingangsstellen genau die Marken in ausreichender Anzahl enthalten, die den Werten (Termen) der Variablen an den entsprechenden Eingangskanten entsprechen.

Die Transition schaltet, indem genau diese Marken aus den Eingangsstellen entfernt werden. Über die Ausgangskanten werden Marken mit den Werten der als Anschriften auftretenden Variablen abgelegt.

Die Suche nach einer Variablenbindung kann folgendermaßen durch einen Prologinterpreter erfolgen. Für alle $s \in \mathrm{S}$ sei $\mathrm{m}(s)$ die Multimenge der Terme (Marken), die auf s liegen. Die aktuelle Markierung des Netzes erzeugt Fakten der Form $\texttt{mark}(s, m)$ für jede Markierung $m \in_{\mathrm{M}} \mathrm{m}(s)$. Ein Beispiel für solch ein Fakt wäre `mark(incoming_orders,order(frank,mainBoard))`. Die Fakten stellen damit dem Prolog-Interpreter die aktuelle Belegung der Stellen mit Marken (Termen) zur Verfügung.

Für alle $t \in \mathrm{T}$ sei $\mathrm{p}(t)$ die Transitionsregel in Form einer Prolog-Regel und für alle Kanten $(x, y) \in \mathrm{S} \times \mathrm{T} \cup \mathrm{T} \times \mathrm{S}$ sei $\mathrm{v}(x, y)$ der Name der Variablen an der Kante von x nach y. Die Transitionsregeln werden ergänzt, indem alle Variablen an Eingangskanten an in den entsprechenden Stellen vorhandene Marken gebunden werden. Dafür wird der Transitionsregel für jede Eingangskantenvariable $\mathrm{v}(s, t)$ ein Unterziel $\texttt{mark}(s, v)$ hinzugefügt.

Das PROLOG-Programm, welches für die aktuelle Markierung gelöst werden muß, um Bindungen von aktivierten Transitionen zu finden, setzt sich aus den ergänzten Transitionsregeln und einer Kodierung der aktuelle Markierung nach dem oben genannten Schema zusammen. Jede gültige Variablenbelegung, die vom Prologinterpreter gefunden wird, kann zum Schalten der jeweiligen Transition verwendet werden[7] .

[7]Wenn man Mehrfachkanten zuläßt, muß zusätzlich beachtet werden, daß jede Marke nur einmal konsumiert werden darf.

Wenden wir uns nun den SGML-/XML-Netzen zu, die vor allem durch ihr Anwendungsgebiet E-Commerce für diese Arbeit interessant sind.

3.4.5 SGML- und XML-Netze

Wie in Abschnitt 3.2.3 beschrieben, gibt es im Bereich der Petrinetze die Bestrebung, den Formalismus so zu erweitern, daß komplexe Datenstrukturen einfach beschrieben werden können. Im E-Commerce hat man es häufig mit strukturierten Dokumenten wie Produktbeschreibungen, Bestellformularen und ähnlichem zu tun.

Weitz schlägt in [Weitz 1998a] und [Weitz 1998b] eine auf das Anwendungsgebiet E-Commerce abgestimmte Erweiterung der Petrinetze vor, die er SGML-Netze nennt. Auf diesem Ansatz aufbauend wurden kürzlich XML-Netze vorgestellt ([Lenz und Oberweis 2001]).

SGML und XML ([Oasis-Open 2000]) stellen als Beschreibungssprache für Dokumenttypen informationsorientierte Modellierungstechniken dar, werden in dieser Arbeit aber nicht tiefergehend behandelt und deshalb an dieser Stelle kurz vorgestellt. Für eine Einführung in SGML bzw. XML sei auf [Goldfarb 1991] bzw. [Goldfarb und Prescod 2000] verwiesen.

XML (*extensible markup language*) stellt eine Spezialisierung von SGML dar, die mittlerweile bekannter ist als SGML selbst. XML hat sich insbesondere für Internet-Anwendungen durchgesetzt, da SGML viele für typische Web- und E-Commerce-Anwendungen nicht benötigte Konzepte und Techniken enthält und deshalb als zu komplex angesehen wird ([Goldfarb und Prescod 2000]). Da die von Weitz genutzten Aspekte von SGML (zumindest die in [Weitz 1998b] beschriebenen) vollständig von XML abgedeckt werden, konnte der Ansatz ohne größere Änderungen auf XML übertragen werden ([Lenz und Oberweis 2001]).

XML- bzw. SGML-Dokumente sind streng hierarchisch strukturierte Dokumente. Textblöcke (*character data*, CDATA) können beliebig tief ineinander verschachtelt werden. Die Unterteilung in Textblöcke und weitere Meta-Information zum Text wird durch syntaktisch abgegrenzte Zusatzinformation (*mark up*) ausgedrückt. Die Syntax von XML und SGML sieht vor, daß Meta-Information in spitze Klammern (*kleiner-als-* und *größer-als-*Zeichen) eingeschlossen werden. Befehle werden als *tags* (nicht zu verwechseln mit den *tags* in Feature Structures, siehe dazu Abschnitt 2.3.1) bezeichnet. *Tags* gibt es in der Regel in einer öffnenden und einer schließenden Variante, welche eingeschlossenen Text und Mark-Up zu einem Block zusammenfassen. In einigen Fällen kann in SGML jedoch, im Gegensatz zu XML, das schließende *tag* entfallen. Dies wurde im Entwurf von SGML so vorgesehen, um den Autoren manuell erstellter SGML-Dokumente Schreibarbeit zu ersparen, hat sich aber letztendlich als ein großer Nachteil von SGML erwiesen. SGML-Dokumente können durch diese syntaktische Ungenauigkeit im Gegensatz zu XML-Dokumenten ohne eine Dokumenttyp-Definition (DTD, siehe unten) nicht auf syntaktische Korrektheit überprüft werden. Unter Berücksichtigung einer DTD ist eine Kontrolle der

Syntax zwar möglich, SGML stellt hier aber höhere Anforderungen an einen *Parser* als XML.

Aus SGML stammt das auch in XML anzutreffende Konzept der Dokumenttyp-Definition (*document type definition*, DTD), welche die Struktur von Dokumenten weiter spezialisiert. Dies geschieht, indem angegeben wird,

- mit welchem *tag* das gesamte Dokument beginnt,
- welche *tags* innerhalb welcher anderen in welcher Reihenfolge und Wiederholung auftreten können (kontextfreie Grammatik) und
- welche Attribute jedes *tag* besitzen darf oder muß und welchen Typ diese Attribute haben.

Mithilfe dieser Angaben läßt sich bestimmten, ob ein nach der allgemeinen XML-Syntax *wohlstrukturiertes* Dokument (*wellformed document*) auch der durch die DTD definierten Grammatik entspricht. Ein Dokument, welches die zweite Bedingung erfüllt, wird in der SGML-/XML-Terminologie als *valides* Dokument (*valid document*) bezeichnet.

Definition von SGML-Netzen

SGML-Netze werden in [Weitz 1998b] nicht formal definiert. Mit Bezug auf die SGML-Spezifikation ist dennoch eine semiformale Definition möglich.

SGML-Netze sind Petrinetze mit Inhibitorkanten (siehe Abschnitt 3.2.4), die SGML-Dokumentdefinitionen als Stellentypen, sogenannte Dokumentmuster als Kantenanschriften und SGML-Dokumente als Marken verwenden. Sie gehören damit zur Familie der höheren Petrinetz (siehe Abschnitt 3.2.3). In einer Stelle mit dem Dokumenttyp D dürfen nur SGML-Dokumente als Marken vorkommen, die bezüglich D valide sind. Wann dies der Fall ist, wird durch die SGML-Spezifikation bestimmt.

Dokumentmuster haben den Zweck, einen Musterabgleich *(pattern matching)* zwischen Marke und Muster zu erlauben, um eine bestimmte Struktur von Marken-Dokumenten zu fordern oder auf bestimmte Teile eines Marken-Dokuments zuzugreifen. Ein Dokumentmuster entspricht in seiner baumartigen Struktur einem SGML-Dokument, wobei die Knoten des Baums *tags* entsprechen. In der graphischen Darstellung können die Knoten entweder mit gestrichelter oder durchgezogener Umrandung dargestellt werden, wobei letzteres eine Modifikation am Dokument an der entsprechenden Stelle spezifiziert. Durch verschiedene Operatoren, die in Dokumentmustern auftreten dürfen, können Variablen definiert, Muster negiert und unter bestimmten Ausnahmebedingungen Marken als Ganzes konsumiert werden („Zap“-Operator).

Eine Besonderheit in bezug auf die gewohnte Petrinetz-Semantik stellt die Möglichkeit in SGML-Netzen dar, mit einer Eingangskante lediglich Teile eines

Marken-Dokuments aus diesem zu entfernen. Es wird also nicht eine Marke als ganzes konsumiert, sondern auf den Inhalt der Marke zugegriffen, ein Teil aus dem Dokument herausgegriffen, und der verbleibende Teil des Dokument als Marke in der Eingangsstelle belassen. Dies ist ein für Petrinetze ungewöhliches Verhalten, da sich so das gefärbte Netz nicht mehr direkt in ein ungefärbtes Netz abbilden läßt, das zumindest die kausale Struktur des Systems darstellt. Weiterhin wird es dadurch schwierig, Marken insgesamt zu konsumieren, wofür Weitz den „Zap"-Operator einführt. Dieser Operator erlaubt es, in dem speziellen Fall, daß das letzte Auftreten eines sich wiederholenden Elements konsumiert wurde, doch die gesamte Marke zu konsumieren.

Sowohl in der fehlenden formalen Definition (Wie bestimmt man, wann eine Marke zu einem Muster paßt? Wie genau werden die Variablen belegt?) als auch in dem Schaltverhalten, das gegen die Intuition bei Petrinetzen verstößt, liegt die Kritik an Weitz' Ansatz. Positiv hervorzuheben ist dagegen, daß auf Konzepte und Techniken zurückgegriffen wird, die im Anwendungsgebiet Verwendung finden, in diesem Fall SGML bzw. XML. Auch wird der Einsatz von Petrinetzen zur Prozeßmodellierung grundsätzlich positiv beurteilt.

In [Lenz und Oberweis 2001] wird mit XML-Netzen der Nachfolger der SGML-Netze vorgestellt. Die Vereinheitlichungen und Vereinfachungen von XML gegenüber SGML wirken sich positiv auf die Definition aus. In [Lenz und Oberweis 2001] wird zwar von einer formalen Definition gesprochen, diese wird dort aber nicht gegeben und es wird auch nicht auf entsprechende Literatur verwiesen. Das in [Weitz 1998b] noch recht unklare Konzept der Dokumentmuster wurde aber weiter ausgearbeitet. GXSL (*graphical XML schema language*) stellt, ähnlich zu der Vorgehensweise in der vorliegenden Arbeit, XML-Schemata durch eine graphische Notation dar, die ein in Details angepaßter Ausschnitt von UML ist.

Um Dokumente abfragen und manipulieren zu können, werden Variablen und fest spezifizierte Werte innerhalb von GXSL-Ausdrücken erlaubt. Diese Erweiterung von GXSL wird XManiLa (*XML manipulation language*) genannt. Analog zur hier eingeführten UML-Notation für Feature Structures werden Objekte und Objektsichten oder -muster durch eine sehr ähnliche Notation ausgedrückt.

Zum Hinzufügen und Löschen von Knoten gibt es wie bei SGML-Netzen verschiedene Knotenarten, nur daß die zu verändernden Knoten nicht wie bei SGML-Netzen durchgezogen (statt gestrichelt) dargestellt werden, sondern einen fetten Balken an der linken Seite erhalten.

Die nicht Petrinetz-konforme Verhaltensweise, über Eingangskanten abhängig von Muster und Marke nur Teile einer Marke zu konsumieren und durch Ausgangskanten bereits vorhandene Dokumentenmarken zu manipulieren (anstatt stets neue Marken zu erzeugen), bleibt in XML-Netzen erhalten. Damit bleibt auch die oben genannte Kritik an SGML-Netzen für XML-Netze bestehen. Eine Werkzeugunterstützung ist nach Aussage in [Lenz und Oberweis 2001] in Vorbereitung.

3.5 Werkzeugunterstützung für Referenznetze: RENEW

RENEW (*reference net workshop*, [Kummer und Wienberg 2000]) ist ein Editor und Simulator für höhere Petrinetze. Das Projekt wurde 1998 von Kummer und dem Autoren ins Leben gerufen, um das Paradigma der Netze in Netzen (siehe Abschnitt 3.3) durch ein geeignetes Werkzeug zu unterstützen und steht als Open-Source zur Verfügung. Andere Petrinetzwerkzeuge hatten sich in vorausgehenden Untersuchungen ([Kummer et al. 1998, Kummer et al. 1999]) als zu schwer anpaßbar oder zu unflexibel erwiesen, um erweiterte Formalismen so realisieren zu können, daß sie effektiv einsetzbar sind. Mit der gerade beginnenden Verbreitung von Java ([Sun 2000b]) stand eine relativ einfache Möglichkeit zur Verfügung, ein Werkzeug mit einer graphischen Benutzungsoberfläche plattformübergreifend zu realisieren.

Der erste realisierte Formalismus, für den RENEW im Moment im wesentlichen eingesetzt wird, sind die in Abschnitt 3.3.3 beschriebenen Referenznetze. RENEW stellt den in Abschnitt 3.3.3 erwähnten erweiterten Referenznetz-Formalismus zur Verfügung, in dem neben Netzreferenzen beliebige Java-Objekte als Marken verwendet werden können. In Transitionen können Methoden der gebundenen Java-Objekt-Marken aufgerufen werden. Allein diese Möglichkeiten stellten sich in unseren Untersuchungen als Erweiterung von DESIGN/CPN als machbar, aber aufwendig heraus ([Kummer et al. 1998, Kummer et al. 1999]). RENEW ist selbst in Java implementiert, so daß sich die Verwendung von Java als Beschriftungssprache und Markensemantik wesentlich natürlicher ergibt.

Für Netze in Netzen mit Wertsemantik gibt es bisher keine Werkzeugunterstützung, obwohl man diese in gewissem Sinne als Spezialfall der Referenznetze betrachten kann. Für den Sonderfall, daß jedes Markennetz nur genau einmal referenziert wird, gibt es keinen Unterschied zwischen Wert- und Referenzsemantik. Um Valks in Abschnitt 3.3.1 beschriebene Semantik der elementaren Objektsysteme zu erreichen, muß zusätzlich das Konzept der Prozeßmarkierung und die Unifikation von Prozessen implementiert werden. Wie in Abschnitt 4.3 gezeigt wird, können die in dieser Arbeit entwickelten FSNets und die in Abschnitt 4.5 vorgestellte Erweiterung von RENEW auf FSNets hier einen Beitrag leisten. Mit dieser Erweiterung ist außerdem erstmals gezeigt worden, daß sich die offene Architektur von RENEW tatsächlich für substantiell andere höhere Petrinetzformalismen eignet.

Im folgenden wird ein kurzer Überblick über die Eigenschaften und Möglichkeiten von Renew gegeben. Für Details sei auf die Web-Site mit diverser Dokumentation ([Kummer und Wienberg 2000]) und verschiedene Literatur verwiesen ([Kummer 1998, Kummer 1999a, Kummer et al. 2000a, Kummer 2001, Valk 1999, Kummer et al. 2001]).

3.5.1 Überblick

RENEW besteht aus einer graphischen Oberfläche, mit der höhere Petrinetze erstellt und bearbeitet werden können, sowie einer Ausführungsmaschine (in der Petrinetz-Welt oft als „Simulator“ bezeichnet), die Referenznetze interpretieren und mit Java in Verbindung bringen kann. Die Verknüpfung zwischen graphischer Oberfläche und Ausführungsmaschine besteht in zweierlei Hinsicht. Zum einen existiert ein Netzcompiler, der graphisch eingegebene Netze in ein Format übersetzt, das von der Maschine interpretiert werden kann. Zum anderen gibt es eine Schnittstelle der Ausführungsmaschine, um eine graphische Animation von Netzausführungen zu erlauben. Diese Schnittstellen wurden so ausgearbeitet, daß sowohl die graphische Oberfläche als auch die Ausführungsmaschine leicht unabhängig voneinander ersetzt oder modifiziert werden können ([Kummer et al. 2001]).

Das Bedienkonzept der graphischen Oberfläche basiert auf dem von bekannten Zeichenwerkzeugen, da die Gestaltung bei einer graphischen Technik wie Petrinetzen einen wichtigen Aspekt des Werkzeugs darstellt. Verschiedene Werkzeuge dienen zum Erstellen und Modifizieren verschiedener Netzelemente oder graphischer Elemente, die zur Illustration von Modellen hinzugefügt werden können. Die Werkzeuge sind auf die Darstellung und Bearbeitung von Graphen optimiert, um Petrinetze gut zu unterstützen. Dabei leistet das JHotDraw-Rahmenwerk von Gamma ([Gamma 1998]), auf dem RENEW aufbaut, wertvolle Dienste.

Da ein Referenznetzsystem aus einer Menge von Netzen besteht und auch andere Petrinetzformalismen durch Verfeinerung oder ähnliche Konzepte die Bearbeitung mehrerer Netze erfordern, gibt es in RENEW ein Fenster für jedes Netz und ein globales Werkzeugfenster. Nur im globalen Fenster steht ein Menü zur Verfügung, um beispielsweise Netze zu laden und zu speichern oder Attribute von Zeichenobjekten zu ändern.

Während der Simulation können als Besonderheit von Referenznetzen für ein Netz mehrere Fenster geöffnet sein, um verschiedene Netzexemplare (siehe Abschnitt 3.3.3) desselben Netzes anzuzeigen. In Ausführung befindliche Netze werden farbig unterlegt, um eine Verwechslung mit der Netzschablone (dem nicht-instantiierten Netz) zu vermeiden.

Im folgenden Abschnitt wird auf eine Erweiterung von RENEW eingegangen, welche die Einsetzbarkeit von Referenznetzen im Bereich des Workflow-Management stark verbessert.

3.5.2 Workflow-Erweiterungen von RENEW

Wie in Abschnitt 3.4 ausgeführt, eignen sich Petrinetze im allgemeinen und Referenznetze im speziellen zur Modellierung, Simulation und Ausführung (*enactment*, [Aalst et al. 1999]) von Geschäftsprozessen und Workflows. Um das Werkzeug RENEW für Workflow-Anwendungen einsetzten zu können, bedarf es aber einiger Erwei-

terungen, welche die Realisierung konkreter Workflow-Anwendungen unterstützen. RENEW stellt ein generisches Werkzeug dar, das für spezielle Anwendungsgebiete parametrisiert werden muß. Weiterhin sind die Anforderungen an Workflow-Management-Systeme gestiegen, so daß eine einfache Ausführung von Aktivitäten nach kausalen Abhängigkeiten nicht mehr als ausreichend betrachtet wird.

Da Referenznetze gefärbte Netze als einen Spezialfall abdecken, können die dort existierenden Workflow-Modellierungstechniken direkt übertragen werden. Die Erweiterungen von Referenznetzen sollen aber für diese spezielle Anwendung genutzt werden. Deshalb wurde Renew innerhalb eines Projekts unter Beteiligung von Studenten um speziell auf Workflow ausgerichtete Techniken erweitert.

Es gibt wie oben bereits gesagt eine direkte Anwendung von Netzexemplaren als Workflow-Exemplare. Was bisher aber kaum genutzt wurde, ist die Möglichkeit, Workflow-Exemplare nicht nur von der Systemumgebung, sondern auch innerhalb eines anderen Workflows zu erzeugen. Damit sind mit RENEW flexiblere Workflows möglich, bei denen der aufzurufende Sub-Workflow erst zur Laufzeit ausgewählt wird.

Das wichtigste Ergebnis dieses Projekts stellt aber eine Erweiterung der RENEW-Programmierschnittstelle dar, die es erlaubt, zu überwachen, welche Transitionen in einem laufenden Netzexemplar unter welchen Bindungen aktiviert sind. Damit wird es möglich, nicht nur mit dem Workflow-Netz Aktivitäten anzustoßen (aus Sicht des Workflow-Management-Systems ein *push*-Modell), sondern auch von außen den Zustand des Systems abfragen und dann gezielt Einfluß auf Entscheidungen im Workflow-Netz nehmen zu können (*pull*-Modell).

Wenn ein Workflow-Netz eine Anwendung oder eine sonstige Aktivität starten soll, so kann dies über die in Referenznetze integrierten Java-Anschriften geschehen. Soll dagegen das Workflow-Netz auf ein externes Signal warten, können Zielkanäle eingesetzt werden, die in RENEW nicht nur von einem anderen Netz, sondern auch von Java-Code aus aktiviert werden können. Die oben erwähnte Introspektion laufender Netze ermöglicht die Implementierung einer Client/Server-Schnittstelle, die einen Workflow-Klienten über mögliche erwartete externe Signale informiert. Dadurch hat der Workflow-Klient im Sinne eines *pull*-Modells die Auswahl zwischen verschiedenen Bindungen, die im Netz geschaltet werden können. Die Schnittstelle zwischen Klient und Server wurde im Projekt über Java-RMI als entfernte Kommunikation realisiert.

Diese generelle Architektur wurde innerhalb des erwähnten Projekts weiter spezialisiert, um die im Bereich des Workflow üblichen Rollen und Aufgabentypen zu unterstützen. Für diese Bereiche werden in der vorliegenden Arbeit in Kapitel 5 eigene Konzepte vorgeschlagen, so daß auf diese Erweiterungen nicht näher eingegangen werden soll. Festzuhalten bleibt also die Möglichkeit, durch externe und entfernte Klienten die Aktivierung von Zielkanälen in einem Netzexemplar abfragen zu können und bei einer entsprechenden Auswahl des Klienten das Netzexemplar weiterschalten zu können.

Kapitel 4

Feature-Structure-Netze: eine informations- und prozeßorientierte Modellierungstechnik

Petrinetze sind zur Modellierung von Prozessen allgemein anerkannt (siehe dazu unter anderem [Reisig 1986], [Jessen und Valk 1987], [Baumgarten 1990], [Jensen 1992]). Sobald aber Daten oder Information einen gewissen Stellenwert erreichen, eignen sie sich bisher nur bedingt zur Erstellung verständlicher, übersichtlicher Modelle. In der Petrinetzgemeinschaft gibt es deshalb zur Zeit die Bestrebung, höhere Petrinetze zu definieren, die präziser und mächtiger als die gefärbten Petrinetze ([Jensen 1992]), jedoch einfacher zu verstehen und zu handhaben sind als die Algebraischen Petrinetze ([Reisig 1990]). Zu nennen sind hier [Oberweis et al. 1994, Jeffrey et al. 1996, Weitz 1998b, Lenz und Oberweis 2001] und diverse Ansätze für objektorientierte Petrinetze [Battiston et al. 1991, Buchs und Guelfi 1991a, Lakos 1995, Moldt 1996, Aoumeur und Saake 1999]. Gerade in den verschiedenen Ansätzen für OO-Petrinetze wird deutlich, daß eine Verbindung von objekt- und prozeßorientierter Modellierung heutzutage zwar erstrebt wird, aber noch nicht vollkommen verstanden ist. Relationale und objektorientierte Techniken eignen sich weiterhin nur bedingt zur Darstellung von Information und Wissen, wie es beispielsweise zur Modellierung von Agenten benötigt wird ([Kinny et al. 1996]).

Aus dieser Motivation sind die Feature-Structure-Netze (*feature-structure-nets*, FSNets) entstanden, die in diesem Kapitel als neuer und eigenständiger Beitrag vorgestellt werden. Feature Structures und die dahinterstehende Typmodellierung dienen wie in Kapitel 2 vorgestellt der intuitiven und dennoch ausdrucksstarken Modellierung von Information, Daten und Wissen. Feature Structures allein sind eine rein deklarative Technik, die nicht für die Beschreibung von Abläufen und imperativen

Problemstellungen geeignet ist, wenn sich auch Regeln und, wie in Abschnitt 4.2.8 gezeigt wird, sogar Prozesse mit ihnen darstellen lassen. Deshalb werden Petrinetze zur Beschreibung von Kausalität und Nebenläufigkeit, kurz für den prozeßorientierten Teil der Modellierung herangezogen. Konkret werden Feature Structures als Marken und als Stellen-, Kanten- und Transitionsanschriften in Petrinetzen benutzt.

Es wird nicht nur eine Notation und natürlichsprachliche Semantik gegeben, die anhand von Beispielen motiviert wird, sondern FSNets werden auch auf eine solide formale Grundlage gestellt. In Abschnitt 4.1 wird eine formale Definition der Struktur von sogenannten Basis-FSNets gegeben, die zunächst ausreicht, um eine graphische Notation einzuführen. Feature Structures und Petrinetze sind formal definiert, so daß diese zur Definition von FSNets herangezogen werden können. Für die folgenden formalen Untersuchungen wird in Abschnitt 4.2 eine beschränkte Version des Basis-FSNet, das elementare FSNet, eingeführt. Für elementare FSNets kann man viele Konstrukte aus der Petrinetztheorie wie eine Markierung, die Wirkung einer Transition und sogar Prozesse mit Feature Structures darstellen. So kann für elementare FSNets eine Prozeßsemantik auf der Basis von Feature Structures definiert werden. Mit dieser Definition eröffnet sich auch die Möglichkeit, elementare Objektsysteme als FSNets zu interpretieren, da Systemnetze Prozesse als Marken besitzen, die nun als Feature Structures dargestellt werden können. Valk beschränkt sich bei den Netzen in Netzen bisher auf ungefärbte Netze, da sich die Unifikation gefärbter Prozesse als schwierig erwiesen hat. Durch die einheitliche Darstellung von Marken und Prozessen als Feature Structures können in Abschnitt 4.3 in diesem Bereich neue Ergebnisse erzielt werden. Im Zuge der Unterstützung durch Werkzeuge haben sich einige praxisorientierte Erweiterungen für FSNets gegenüber dem Basisformalismus ergeben, was zum Modell der höheren FSNets (HFSNets) führt. Für die Erweiterungen werden keine formalen Definitionen, wohl aber genaue Syntax und (zumindest natürlichsprachliche) Semantik in Abschnitt 4.4 gegeben. Zuletzt wird in diesem Kapitel in Abschnitt 4.5 auf die Werkzeugunterstützung für FSNets, die in das bereits vorgestellte Petrinetzwerkzeug Renew integriert wurde, hingewiesen.

4.1 Basis-Feature-Structure-Netze

Wir beginnen die Darstellung der FSNets mit dem Modell der Basis-FSNets, das die wesentlichen Konzepte von Feature Structures als Marken in höheren Petrinetzen verdeutlicht und ohne zu großen technischen Aufwand formal definierbar ist. Während elementare EFNets Einschränkungen vornehmen, um über eine Definition hinaus formale Ergebnisse zu erzielen, erweitern höhere FSNets den Basisformalismus um Konstrukte, die in dieser Arbeit nicht formal definiert werden.

Ein entscheidender Vorteil von Petrinetzen als Modellierungstechnik ist ihre graphische Repräsentation. Um die Regeln für eine anschauliche Repräsentation eines FSNet genau angeben zu können, muß aber zumindest die Struktur des Netzes be-

reits formal definiert sein. Nachdem im nächsten Abschnitt 4.1.1 diese Definition gegeben wird, wird in Abschnitt 4.1.2 in mehreren Schritten eine Standardnotation für Basis-FSNets hergeleitet, die sich exakt auf die formale Definition abbilden läßt. Abschnitt 4.1.3 definiert eine operationale Semantik für Basis-FSNets durch die Definition von Aktivierung und Schalten von Transitionen.

4.1.1 Formale Definition von Basis-FSNets

Ein Basis-FSNet entspricht einem gefärbten Petrinetz (siehe Abschnitt 3.2.3) mit einem Typsystem, bei dem für Stellentypen, Transitionsregeln und Anfangsmarkierungen Feature Structures über diesem Typsystem genutzt werden. Den Kanten werden Pfade zugeordnet, die sich auf die Transitionsregel beziehen.

Definition 4.1 [Basis-FSNet]
Ein Basis-Feature-Structure-Netz (Basis-FSNet) ist ein Tupel $BFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{s_{type}}, \mathrm{l}, \mathrm{r}, \mathrm{m_0} \rangle$ aus folgenden Komponenten:

(a) endlichen Mengen von *Stellen* S und *Transitionen* T mit $\mathrm{S} \cap \mathrm{T} = \emptyset$,

(b) einem *Typsystem* $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ (Def. 2.4),

(c) einer *Stellentypisierungsfunktion* $\mathrm{s_{type}} : \mathrm{S} \to \mathrm{WFS_{TS}}$, die jeder Stelle eine wohltypisierte Feature Structure als Typ zuweist,

(d) einer *Kantenanschriftfunktion* $\mathrm{l} : (\mathrm{S} \times \mathrm{T}) \cup (\mathrm{T} \times \mathrm{S}) \to \mathcal{P}_{\mathrm{M}}(\mathrm{Feat}^*)$, die jeder Kante eine Multimenge von Pfaden zuordnet, wobei sich die Flußrelation $\mathrm{F} \subseteq \mathrm{S} \times \mathrm{T} \cup \mathrm{T} \times \mathrm{S}$ aus der Def. von Petrinetzen ergibt als $(x, y) \in \mathrm{F} \Leftrightarrow \mathrm{l}(x, y) \neq \emptyset_{\mathrm{M}}$,

(e) einer *Transitionsregelfunktion* $\mathrm{r} : \mathrm{T} \to \mathrm{WFS_{TS}}$, die jeder Transition eine wohltypisierte Feature Structure als Schaltregel zuweist und

(f) einer Anfangsmarkierungsfunktion $\mathrm{m_0} : \mathrm{S} \to \mathcal{P}_{\mathrm{M}}(\mathrm{WFS_{TS}})$, die jeder Stelle eine Multimenge von wohltypisierten Feature Structures zuordnet.

Um ein (statisch) wohltypisiertes Basis-FSNet zu erhalten, müssen weiterhin die Stellen-, Kanten- und Transitionsanschriften mit den Stellentypen konform gehen. Dazu gelte

(g) für alle $s \in \mathrm{S}, F \in_{\mathrm{M}} \mathrm{m_0}(s) : \mathrm{s_{type}}(s) \sqsubseteq F$
(die Anfangsmarkierung ist wohltypisiert) und

(h) für alle $t \in \mathrm{T}, s \in \mathrm{S}, \pi \in_{\mathrm{M}} \mathrm{l}(s, t) \cup_{\mathrm{M}} \mathrm{l}(t, s) : \mathrm{s_{type}}(s) \sqsubseteq \mathrm{r}(t)@\pi$
(die Kantenanschriften und Transitionsregeln sind zueinander und zu den Stellentypen kompatibel). ◊

4.1.2 Graphische Notation von Basis-FSNets

Bei der graphischen Notation für FSNets ist es sinnvoll, sich an die Schreibweisen für andere High-Level-Petrinetze anzulehnen. Als Gemeinsamkeiten läßt sich feststellen, daß Stellen als Ellipsen, Transitionen als Rechtecke und die Elemente der Flußrelation als gerichtete Kanten notiert werden. Außerdem werden Stellentypen, Kantenanschriften und Transitionsregeln wie üblich durch textuelle Anschriften in unmittelbarer Nähe der graphischen Elemente repräsentiert.

Sei $BFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{s_{type}}, \mathrm{l}, \mathrm{r}, \mathrm{m_0} \rangle$ ein Basis-FSNet. In Abschnitt 2.1.4 wurde eine UML-basierte graphische Notation für Konzeptsysteme eingeführt, die hier wiederverwendet werden soll, um ein Konzeptsystem zu spezifizieren, aus dem nach Abschnitt 2.1.3 direkt das Typsystem TS abgeleitet werden kann.

Die Stellen S, Transitionen T und die (implizite) Flußrelation F werden dem Standard entsprechend dargestellt. Die Stellentypisierungsfunktion $\mathrm{s_{type}}$ wird in AVM-Notation als Stellenanschrift dargestellt, wobei hier wie überall sonst die in Abschnitt 2.3.5 vorgestellten Abkürzungen benutzt werden können. Die Kantenanschriftfunktion l wird als textuelle Anschrift an den Kanten dargestellt, wobei die Features eines Pfads durch einen Doppelpunkt getrennt werden und eine Multimenge von Pfaden durch die Aufzählung der Elemente mit ihrer Multiplizität analog zu Definition A.11 dargestellt wird. Eine Multiplizität von 1 kann entsprechend weggelassen werden. Als Trennzeichen der Elemente wird das Komma verwendet. Der leere Pfad ε, der auf den Wurzelknoten der Transitionsregel verweist, kann weggelassen werden, wenn eine Multiplizität angegeben wurde oder wenn er das einzige Element der Multimenge ist, was zu einer leeren Anschrift führt. Also sind feat, feat1 : feat2, 2′feat1 : feat2 und feat, 2′feat1 : feat2 und die leere Zeichenkette erlaubte Kantenanschriften. Transitionsregeln aus r werden als Feature Structures in AVM-Notation innerhalb der entsprechenden Transition notiert. Die Anfangsmarkierung besteht aus einer Multimenge von Feature Structures pro Stelle, von der jedes Element in der in Abschnitt 2.3.1 eingeführten AVM-Notation in die entsprechende Stelle geschrieben wird. Multimengen über AVMs werden analog zu Multimengen über Pfaden notiert.

Die Ableitung eines formalen FSNet aus der graphischen Darstellung sei anhand eines Beispiels veranschaulicht.

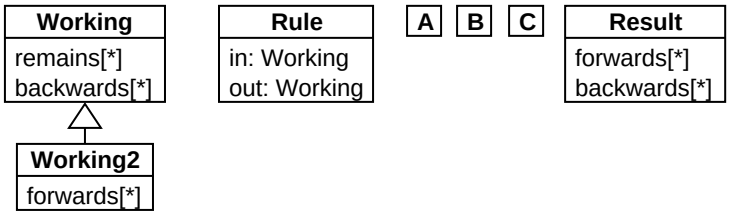

Abbildung 4.1: Das Konzeptsystem, das den folgenden FSNets zugrundeliegt.

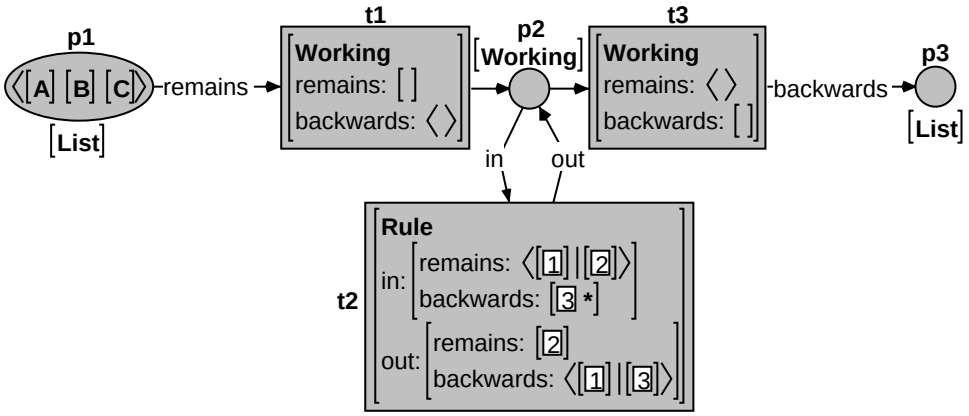

Abbildung 4.2: Ein Beispiel für die erste Variante der Notation von FSNets. Das FSNet kehrt eine gegebene Liste um.

Abbildung 4.1 zeigt das Konzeptsystem, aus dem das allen folgenden FSNets zugrundeliegende Typsystem abgeleitet wird. Den Ausgangspunkt bildet das FSNet in Abbildung 4.2, das die Reihenfolge der Elemente einer gegebenen Liste umkehrt. Wie dieses FSNet funktioniert, das heißt welche operationale Semantik es erhält, wird erst in den nächsten Abschnitten formal definiert. Betrachten wir für den Moment nur die Struktur des Netzes.

Es soll ein Basis-FSNet $BFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{s_{type}}, \mathrm{l}, \mathrm{r}, \mathrm{m_0} \rangle$ aus den Abbildungen 4.1 und 4.2 rekonstruiert werden. Aus Abbildung 4.1 ergibt sich folgendes Konzeptsystem: Die Konzepte Working, Working2, Rule, Result, A, B und C werden mit ihren Feature-Typisierungen der Attribute in, out, remains und backwards und ihren Subtyp-Beziehungen (im wesentlichen Working2 ISA Working) dem in Abschnitt 2.3.5 eingeführten Standard-Konzeptsystem für Listen und Tupel (letztere im Beispiel nicht verwendet) hinzugefügt. Einige der Konzepte werden erst in späteren Versionen des Beispielnetzes benötigt. Aus dem so erhaltenen Konzeptsystem wird nach Abschnitt 2.1.3 das Typsystem TS abgeleitet. In diesem Fall unterscheidet sich dieses nur durch den zusätzlichen Typ $\top$ vom Konzeptsystem, da im Konzeptsystem keine überlappende Vererbung auftritt.

Aus Abbildung 4.2 ergibt sich das eigentliche Basis-FSNet. Die Stellen p1, p2, p3 und die Transitionen t1, t2, t3 führen zu den Mengen $\mathrm{S} = \{\mathsf{p1}, \mathsf{p2}, \mathsf{p3}\}$ und $\mathrm{T} = \{\mathsf{t1}, \mathsf{t2}, \mathsf{t3}\}$. Aus den Kanten läßt sich die Kantenanschriftfunktion l mit $\mathrm{l}(\mathsf{p1}, \mathsf{t1}) = \{\mathsf{remains}\}_\mathrm{M}$, $\mathrm{l}(\mathsf{t1}, \mathsf{p2}) = \{\varepsilon\}_\mathrm{M}$, $\mathrm{l}(\mathsf{p2}, \mathsf{t2}) = \{\mathsf{in}\}_\mathrm{M}$, $\mathrm{l}(\mathsf{t2}, \mathsf{p2}) = \{\mathsf{out}\}_\mathrm{M}$, $\mathrm{l}(\mathsf{p2}, \mathsf{t3}) = \{\varepsilon\}_\mathrm{M}$, $\mathrm{l}(\mathsf{t3}, \mathsf{p3}) = \{\mathsf{backwards}\}_\mathrm{M}$ herleiten. Aus den Anschriften der Stellen ergibt sich $\mathrm{s_{type}}$ als $\mathrm{s_{type}}(\mathsf{p1}) = [\mathsf{List}]$, $\mathrm{s_{type}}(\mathsf{p2}) = [\mathsf{Working}]$ und $\mathrm{s_{type}}(\mathsf{p3}) = [\mathsf{List}]$. Die Anfangsmarkierung betrifft nur die Stelle p1, also ist $\mathrm{m_0}(\mathrm{p1}) = \{\langle [\mathsf{A}][\mathsf{B}][\mathsf{C}] \rangle\}_\mathrm{M}$, die leere Multimenge sonst. Die Regelfunktion r schließlich bildet die drei Transitio-

nen t1, t2 und t3 auf die in ihnen dargestellten Feature Structures ab.

Die Abbildung 4.1 gezeigte Notation orientiert sich sehr nahe an der formalen Definition. Eine flexiblere und kompaktere Notation ist für die Modellierung aber wünschenswert, so daß aus dieser *ersten Variante* im folgenden schrittweise alternative Notationen entwickelt werden, die sich aber alle leicht auf die formale Definition zurückführen lassen.

In Analogie zur AVM-Notation können die Pfade, die in der ersten Notationsversion in den Anschriften der Kanten auftreten, durch Knotenreferenzen ersetzt werden, die auf die entsprechenden Knoten der zugehörigen Transitionsregel verweisen. Der bisher weggelassene Pfad ε braucht nicht durch eine Knotenreferenz ersetzt zu werden, da er eindeutig auf den Wurzelknoten der Transitionsregel verweist. Man erhält für das FSNet aus Abbildung 4.2 als *zweite Variante* die in Abbildung 4.3 gezeigte Notation.

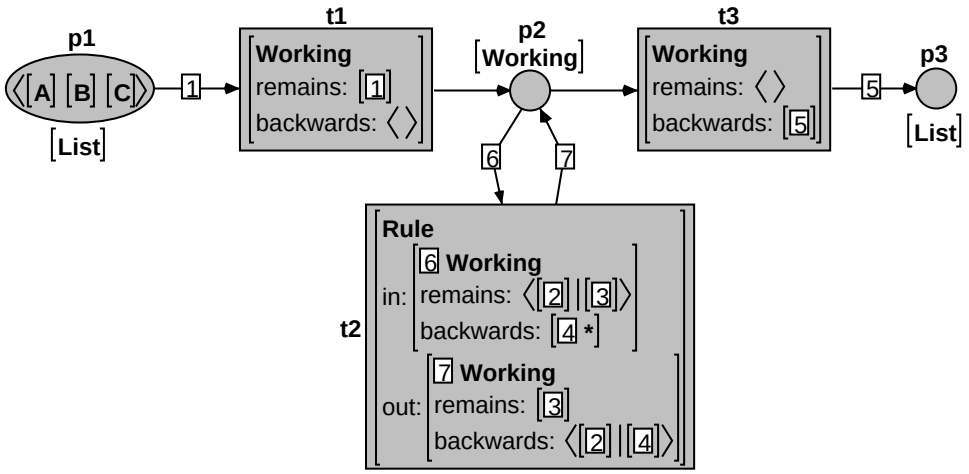

Abbildung 4.3: Zweite Variante: In der Notation aus Abbildung 4.2 wurden Pfade durch Knotenreferenzen ersetzt.

Weiterhin fällt anhand des Beispiels auf, daß auf den Wurzelknoten der Transitionsregel von t2 von keiner Kante aus (weder direkt noch indirekt) verwiesen wird. Nach der im nächsten Abschnitt definierten Schaltregel für Basis-FSNets hat ein solcher Knoten keinen Einfluß auf das Schaltverhalten der Transition und sollte deshalb in einer vereinfachten Notation weggelassen werden. Der Grund für das Vorhandensein solcher Knoten im Formalismus ist, daß die Transitionsregel als zusammenhängende Feature Structure formal besser zu behandeln ist. Auf Knoten der Transitionsregel wird im Modell durch Pfade zugegriffen, die relativ zum Wurzelknoten dargestellt werden. Von den Kanten aus nicht erreichbare Knoten können demnach nicht im Modell, sondern nur in der Notation weggelassen werden. Aus

einer vereinfachten Notation kann ein semantisch identisches Modell einfach durch einen künstlichen Wurzelknoten wiederhergestellt werden.

Wir erlauben also, durch Kantenanschriften nicht erreichbare Knoten der Transitionsregel wegzulassen und die Transitionsregel als *mehrere* AVMs zu notieren. Zu beachten ist, daß sich diese AVMs im Gegensatz zu allgemeinen Notation (siehe Abschnitt 2.3.1) auf *eine* Feature Structure beziehen und somit durch gleiche Knotenreferenz-Bezeichner Strukturgleichheiten spezifiziert werden. Falls mehrere Feature Structures innerhalb einer Transition stehen, darf keine der Kanten dieser Transition unbeschriftet sein, da sonst nicht klar wäre, auf welchen der Wurzelknoten sich die Kante bezieht. Man erhält die *dritte Variante* der Notation in Abbildung 4.4, in welcher der Wurzelknoten der Transitionsregel von t2 weggefallen ist.

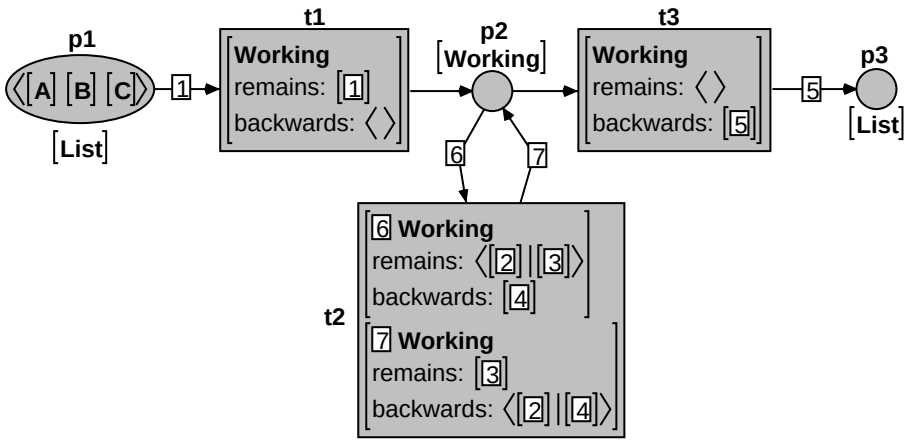

Abbildung 4.4: Dritte Variante: In der Notation aus Abbildung 4.3 wurden von Kanten aus nicht erreichbare Knoten in Transitionsregeln weggelassen.

Nachdem nun eine Transitionsregel in mehrere Feature Structures aufgeteilt wurde, die im Prinzip Mustern für die konsumierten oder erzeugten Marken entsprechen, liegt es nahe, diese Feature Structures direkt an die entsprechenden Kanten zu schreiben, was uns zur *vierten Variante* führt.

Abbildung 4.5 zeigt das bekannte Netz in dieser vierten Notationsvariante. Auf den ersten Blick sieht das Netz übersichtlicher aus. Die Arbeitsweise der Transition t2 ist aber nicht mehr so leicht zu verstehen, da ihre AVM-Kantenanschriften nicht mehr direkt beieinander stehen. Trotzdem ist der Gültigkeitsbereich der in den AVMs vorkommenden Knotenreferenzen nicht eine AVM, sondern die Transition mitsamt ihren Kanten. Dies entspricht dem üblichen Gültigkeitsbereich von Variablen in höheren Netzen wie gefärbten Netzen, Pr/T-Netzen oder auch Referenznetzen, in denen gleiche Bezeichner nur dann gleiche Werte bezeichnen, wenn sie Anschriften derselben Transition oder deren Kanten sind. Obwohl Knotenreferen-

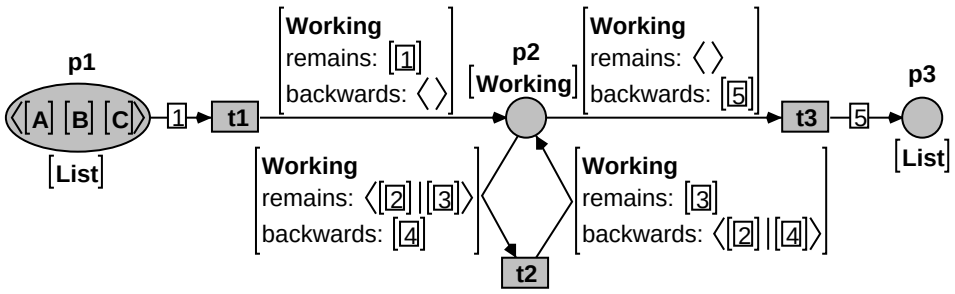

Abbildung 4.5: Vierte Variante: In der Notation aus Abbildung 4.2 wurden die Transitionsregel-AVMs direkt an die entsprechenden Kanten geschrieben.

zen in FSNets also nur lokal zu einer Transition gelten, wurden in den Abbildungen global eindeutige Knotenreferenz-Bezeichner gewählt, um Verwechslungen zu vermeiden.

Der Vorteil der Notation der AVMs an den Kanten ist, daß das Muster, mit dem die durch die Kanten der Transition gebundenen Marken abgeglichen werden, direkt an der jeweiligen Kante zu finden sind. Damit kann das Verständnis der Funktion des Netzes erleichtert werden. Andererseits geht bei komplexeren Regeln wie in t2 eventuell der Zusammenhang der Eingangs- und Ausgangsmuster verloren.

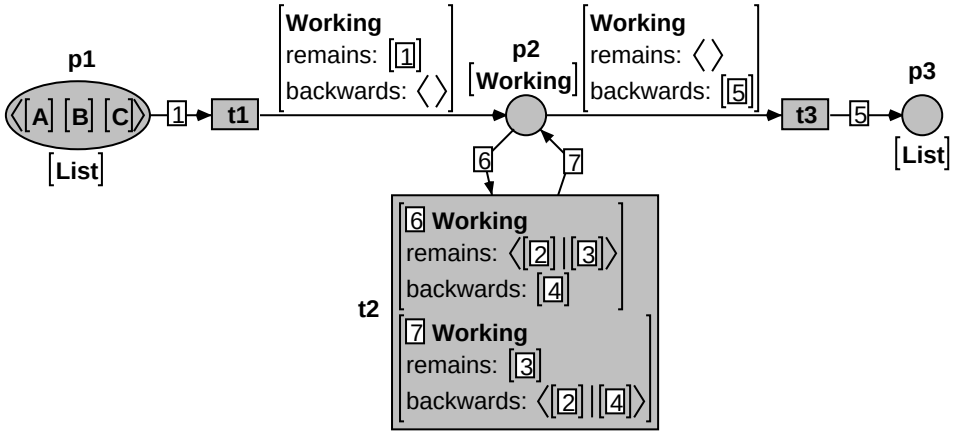

Abbildung 4.6: Fünfte Variante und Standard: Die Notationsmöglichkeiten aus Abbildung 4.4 und Abbildung 4.5 wurden kombiniert.

Als ultimative Notation wird deshalb eine *fünfte Variante* für die Notation von FSNets eingeführt, welche AVMs an Kanten *und* Transitionen erlaubt. Damit stellen sich die Varianten zwei bis vier als Sonderfälle dieser fünften dar, weshalb wir diese

im folgenden als Standardnotation verwenden. In der Standardnotation hat der Modellierer die freie Wahl, an welcher Stelle die Notation der AVM seiner Meinung nach am besten lesbar ist. In den Varianten zwei und drei sind die Knotenreferenzen an den Kanten triviale AVMs aus einem Wurzelknoten, der auf einen an anderer Stelle definierten Knoten verweist. Als AVM notiert müßte die Knotenreferenz von eckigen Klammern umschlossen werden. Um diese Notationen syntaktisch genau so in die Standardnotation abbilden zu können, erlauben wir als Sonderfall, bei AVMs, die nur aus einer Knotenreferenz bestehen, die eckigen Klammern wegzulassen. Notation vier ergibt sich, indem in der Standardnotation keine Transitionsregeln genutzt werden.

Sämtliche weiteren FSNets dieser Arbeit können nach der Standardnotation interpretiert werden. Weitere Notationsvarianten ergeben sich nicht aus einer anderen Darstellung der FSNets, sondern aus den alternativen Notationen für Feature Structures. In Kapitel 5 wird vor allem die in Abschnitt 2.3.5 eingeführte UML-Notation für Feature Structures intensiv eingesetzt, da UML in der Praxis bekannter ist und UML-Feature-Structure als Netzanschriften leichter lesbar sind. Im folgenden werden auch Varianten vorgeschlagen, welche die Graphnotation für Feature Structures nutzen, jedoch in dieser Arbeit nicht weiter verwendet werden. Als Exkurs und Ausblick auf alternative Darstellungen sind die letztgenannten Varianten aber eine interessante Diskussionsgrundlage.

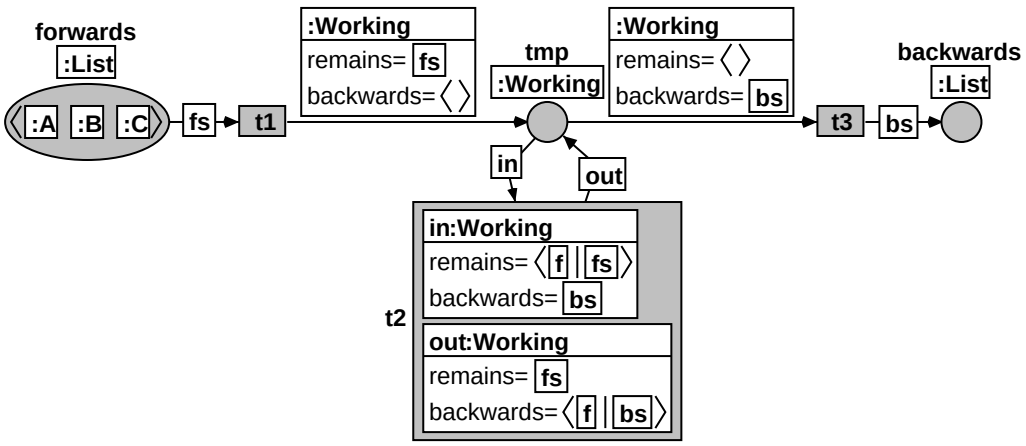

Abbildung 4.7: Das FSNet in Standardnotation wie in Abbildung 4.6, jedoch mit UML-Notation für Feature Structures und alphanumerischen Knotenreferenzen.

Abbildung 4.7 zeigt das FSNet aus Abbildung 4.6 mit UML-Notation für Feature Structures. Für die Knotenreferenzen wurden alphanumerischen Bezeichner gewählt und die Stellen sind aussagekräftiger benannt. Diese Darstellung ist nicht nur für UML-geschulte Modellierer leichter lesbar, sondern betont generell stärker den Bezug

von FSNets zur Objektorientierung, während die Darstellung von Feature Structures als AVMs auf eine Anwendung in der KI hindeutet. Entsprechend wird im Anwendungsteil für das Agentenbeispiel (Abschnitt 5.3) die AVM-Notation, für die anderen Beispiele in Kapitel 5 aber die UML-Notation von Feature Structures genutzt. Im Werkzeug FSRenew (siehe Abschnitt 4.5) kann zwischen beiden Darstellungen der Standard-Notation umgeschaltet werden.

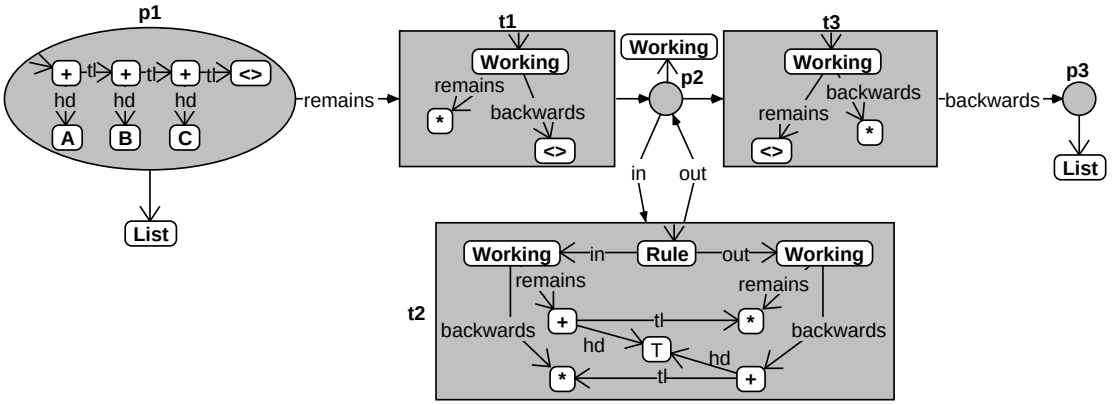

Abbildung 4.8: Das FSNet aus Abbildung 4.2 mit Feature Structures in Graphnotation.

Die weiteren Notationsvarianten greifen auf die Darstellung von Feature Structures als Graphen zurück. Abbildung 4.8 führt diese Notation anhand des FSNet aus Abbildung 4.2 durch. Die Kante, die den Wurzelknoten der Feature Structure auszeichnet, wird hier gleichzeitig verwendet, um die Feature Structure dem entsprechenden Netzelement zuzuordnen. So sind Transitionen mit der Wurzel ihrer Transitionsregel und Stellen mit den Wurzeln ihrer Typen (außerhalb der Stelle) und ihrer Anfangsmarkierungen (innerhalb der Stelle) verbunden.

Auch für diese Graphnotation können analoge Vereinfachungen durchgeführt werden: Von Kanten nicht erreichbare Knoten von Transitionsregeln (wie der Wurzelknoten von t2) können weggelassen werden und Kanten werden nicht mit Pfaden beschriftet, sondern zeigen direkt auf den Knoten der Transitionsregel. Da in der Graphnotation von Feature Structures keine Knotenreferenzen benötigt werden, sondern jeder Knoten genau einmal dargestellt wird und im Falle von Knotenreferenzen in der AVM-Notation mehrere eingehende Kanten besitzt, wählen wir als Notation, daß Kanten die Stellen direkt mit den entsprechenden Knoten der Transitionsregel verbinden. Damit erhalten wir eine *siebte Notationsvariante* in Abbildung 4.9. Alternativ könnten Netzkanten und Feature-Structure-Knoten mit einer weiteren Kantenart verbunden werden, um die Zuordnung von Kanten zu Substrukturen darzustellen, diese Notation wird hier aber nicht weiterverfolgt.

Diese Art der Darstellung führt einerseits dazu, daß der graphische Aspekt der

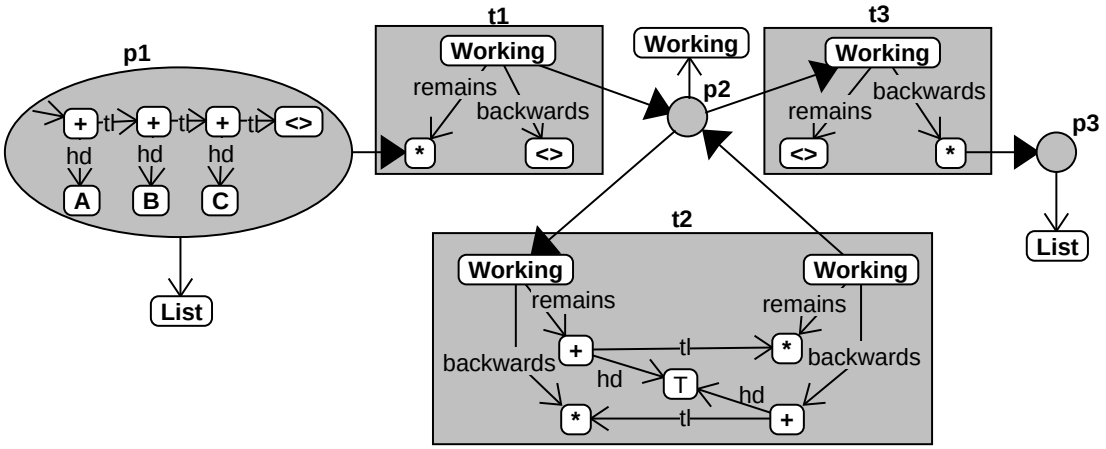

Abbildung 4.9: Die Graphnotation aus Abbildung 4.8 in vereinfachter Notation.

Modelle noch stärker betont wird. Andererseits nimmt die Komplexität der graphischen Darstellung zu, da wir es hier mit drei Knoten- und (mindestens) zwei Kantenarten[1] zu tun haben. Weiterhin kann die ähnliche Darstellung der informations- (Feature Structures) und prozeßorientierten Modellierungsapekte (Petrinetze) durch Graphen schwer auseinanderzuhalten sein und damit für Verwirrung sorgen.

Rein graphische Repräsentationen sind oft gewöhnungsbedürftiger, da sich ihre Bedeutung nicht wie bei textuellen Repräsentationen aus Schlüssel- oder Befehlsworten erschließen läßt. Mit etwas Übung sind graphische Repräsentationen aber meist schneller zu erfassen als textuelle, vor allem, was übergreifende Zusammenhänge betrifft. So erkennt man das symmetrische Verhalten der Transitionen des Netzes in Abbildung 4.9 auf einen Blick. Vor allem an Transition t2 ist hier deutlich der Charakter einer *Graphtransformationsregel* zu erkennen.

Ob sich diese Darstellung für größere Beispiele eignet, darf bezweifelt werden, sollte aber aufgrund des konsequent graphischen Charakters durchaus weiterverfolgt werden. Weitere Optimierungen in der graphischen Darstellung sind dabei denkbar. Eine Möglichkeit dafür stellt die in Abschnitt 2.3.1 vorgestellte Mischnotation dar, in der je nach Situation Feature Structures als Graphen oder AVMs dargestellt werden. Wird diese Darstellung auf die UML-Notation für Feature Structures übertragen, ergeben sich Objektgraphen wie in Abbildung 2.9, die (einige) Attribute als Assoziationen darstellen. Wenn die Transitionsregeln auf komplexen Kollaborationen basieren (siehe Abschnitt 2.2.2), kann eine Darstellung besseren Lesbarkeit führen.

[1] Die Zahl der Kantenarten hängt davon ab, ob man Ein- und Ausgangskanten in Petrinetzen als dieselbe Kantenart ansieht. Auch die Feature-Kanten können noch weiter danach unterschieden werden, ob sie von einem Netzelement auf einen Feature-Structure-Knoten zeigen (Anfangsmarkierung bzw. Transitionsregel) oder zwei Feature-Structure-Knoten verbinden.

Nachdem die graphische Notation von FSNets ausgiebig dargestellt und diskutiert wurde, wenden wir uns nun ihrer Funktionsweise zu.

4.1.3 Eine Schaltfolgensemantik für Basis-FSNets

Um für FSNets eine einfache operationale Semantik anzugeben, werden in diesem Abschnitt die für Petrinetze üblichen Begriffe der Aktivierung und des Schaltens einer Transition definiert (vgl. Abschnitt 3.2.2). Damit erhält man eine Schaltfolgensemantik (*interleaving semantics*), die nebenläufige Schaltfolgen durch deren mögliche Sequenzialisierungen repräsentiert und damit nur eine eingeschränkte Betrachtung der Nebenläufigkeit und Kausalität des modellierten Systems erlaubt. Um FSNets eine formale operationale Semantik zu geben, reicht die Definition von Aktivierung und Schalten aber aus. Für elementare FSNets (siehe Abschnitt 4.2) wird dagegen sowohl eine Schrittsemantik als auch eine Prozeßsemantik definiert.

Zunächst benötigt man eine Notation für die aktuelle Markierung eines in Ausführung befindlichen FSNet. Dazu definieren wir eine Markierung analog zu der Anfangsmarkierung eines FSNet aus Def. 4.1.

Definition 4.2 [Basis-FSNet-Markierung]
Eine Markierung m eines Basis-FSNet $BFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{s_{type}}, \mathrm{l}, \mathrm{r}, \mathrm{m_0} \rangle$ ist eine Funktion $\mathrm{m} : \mathrm{S} \rightarrow \mathcal{P}_{\mathrm{M}}(\mathrm{WFS_{TS}})$, die jeder Stelle eine Multimenge von wohltypisierten Feature Structures zuordnet, so daß gilt: $\forall s \in \mathrm{S}, F \in_{\mathrm{M}} \mathrm{m}(FS) : \mathrm{s_{type}}(s) \sqsubseteq F$ (die Marken entsprechen den Stellentypen). ♢

In gefärbten Netzen spricht man bei Aktivierung und Schalten einer Transition von einem Schaltmodus oder, analog zu der „Farbe“ einer Marke, von der Farbe eines Transitionsvorkommens. Für die Definition eines Schaltmodus tritt in gefärbten Netzen der Begriff der Variablenbindung auf. In FSNets gibt es nun zunächst keine Variablen, der Schaltmodus kann vielmehr elegant wiederum mit Hilfe einer Feature Structure dargestellt werden.

Es soll zunächst eine algebraische Definition des Schaltmodus als eine Feature Structure, die bestimmte Eigenschaften erfüllt, gegeben werden. Dabei wird als erstes eine Markenbindung definiert, die unabhängig von der Transitionsregel mögliche Bindungen von Marken an die Eingangskanten einer Transition beschreibt.

Definition 4.3 [Basis-FSNet-Markenbindung]
Sei $BFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{s_{type}}, \mathrm{l}, \mathrm{r}, \mathrm{m_0} \rangle$ ein Basis-FSNet, m eine Markierung von $BFSN$ und $t \in \mathrm{T}$ eine Transition. Eine *Markenbindung* von t in der Markierung m von $BFSN$ ist eine Funktion $\mathrm{bind} : \mathrm{S} \times \mathrm{Feat}^* \rightarrow \mathcal{P}_{\mathrm{M}}(\mathrm{WFS_{TS}})$, die Paaren aus Stellen und Pfaden Multimengen von Feature Structures zuordnet, für die gilt:

(a) $\forall s \in \mathrm{S}, \pi \in \mathrm{Feat}^* : |\mathrm{bind}(\mathrm{s}, \pi)|_{\mathrm{M}} = \mathrm{l}(\mathrm{s}, \mathrm{t})(\pi)$
(es werden von jeder Stelle unter jedem Pfad genau so viele Marken gebunden, wie die Kantenanschriften fordern) und

(b) $\forall s \in \mathrm{S} : \bigcup_{\mathrm{M}}\{\mathrm{bind}(\mathrm{s},\pi) \mid \pi \in \mathrm{Feat}^*\}_{\mathrm{M}} \subseteq_{\mathrm{M}} \mathrm{m(s)}$
(es werden nur in den entsprechenden Stellen vorhandene Marken gebunden).

$\Diamond$

Nun ergibt sich der eigentliche Schaltmodus als eine Feature Structure, welche die Transitionsregel um genau die Information spezialisiert, die sich aus einer Markenbindung ergibt.

Definition 4.4 [Basis-FSNet-Schaltmodus]
Sei $BFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{s_{type}}, \mathrm{l}, \mathrm{r}, \mathrm{m_0}\rangle$ ein Basis-FSNet, m eine Markierung von $BFSN$ und $t \in \mathrm{T}$ eine Transition. Eine Feature Structure $SMFS \in \mathrm{WFS_{TS}}$ heißt ein *Schaltmodus* von t in m, wenn es eine Markenbindung bind zu t und m gibt, für die gilt:

$$\mathrm{r}(t) \sqcup \bigsqcup\{\pi :: F \mid \pi \in \mathrm{Feat}^*, s \in \mathrm{S}, F \in_{\mathrm{M}} \mathrm{bind}(s,\pi)\} \sim SMFS \qquad \Diamond$$

An dieser Stelle könnte man auch eine schwächere Forderung stellen, indem man die alphabetische Varianz durch eine Subsumption ersetzt und damit auch speziellere Schaltmodi erlaubt. Dies entspricht der Diskussion bei gefärbten Petrinetzen, ob unbelegte Variablen an Ausgangskanten zufällig entsprechend ihres Wertebereichs belegt werden (wie beispielsweise in DESIGN/CPN möglich), oder aber ob sie (wie beispielsweise in RENEW) dazu führen, daß nicht geschaltet werden kann. Da Feature Structures oft (je nach Typsystem) beliebig weiter spezialisierbar sind, scheint eine solche Definition des Schaltmodus, in der Information „hinzuerfunden“ werden kann, aber nicht sinnvoll.

Damit ergibt sich die folgende Definition für Aktivierung und Schalten einer Transition in einem Basis-FSNet.

Definition 4.5 [Aktivierung, Schalten in Basis-FSNets]
Sei $BFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{s_{type}}, \mathrm{l}, \mathrm{r}, \mathrm{m_0}\rangle$ ein Basis-FSNet, $\mathrm{m_1}$, $\mathrm{m_2}$ Markierungen von $BFSN$, $t \in \mathrm{T}$ eine Transition.

(a) t heißt aktiviert (activated, firable), falls es einen Schaltmodus $SMFS$ zu $BFSN$, t und $\mathrm{m_1}$ gibt.

(b) Sei t in $\mathrm{m_1}$ im Schaltmodus $SMFS$ aktiviert und sei bind eine Markenbindung, wie sie nach Definition 4.4 zum Schaltmodus $SMFS$ existiert. t schaltet $\mathrm{m_1}$ zu $\mathrm{m_2}$ im Schaltmodus $SMFS$, symbolisch $\mathrm{m_1} \xrightarrow[\mathrm{bind}]{t} \mathrm{m_2}$ (fires $\mathrm{m_1}$ to $\mathrm{m_2}$), indem $\mathrm{m_2}$ folgendermaßen bestimmt wird:

$$\begin{aligned} \forall s \in \mathrm{S} : \mathrm{m_2}(s) := \mathrm{m_1}(s) \quad & -_{\mathrm{M}} \quad \bigcup_{\mathrm{M}}\{\mathrm{bind}(s,\pi) \mid \pi \in \mathrm{Feat}^*\}_{\mathrm{M}} \\ & \cup_{\mathrm{M}} \quad \bigcup_{\mathrm{M}}\{\mathrm{l}(t,s)(\pi)'(SMFS@\pi) \mid \pi \in \mathrm{Feat}^*\}_{\mathrm{M}} \end{aligned}$$

$\Diamond$

Das Schalten einer Transition garantiert eine wohldefinierte Folgemarkierung, da die Definition 4.3 der Markenbindung in Punkt (b) dafür sorgt, daß der negative Teilausdruck in der Definition der Folgemarkierung m_2 eine Multimengen-Teilmenge von m_1 ist. Weiterhin können nur wohltypisierte Marken abgelegt werden, da die statische Typisierung fordert, daß für alle $t \in T, s \in S, \pi \in_M l(s,t) \cup_M l(t,s)$: $s_{type}(s) \sqsubseteq r(t)@\pi$ und der Schaltmodus die Transitionsregel spezialisiert.

Der Unterschied zwischen Aktivierung und Schalten bei FSNets und anderen höheren Petrinetzen liegt vor allem in der Behandlung von Gleichheit und Identität. Während in allen dem Autoren bekannten höheren Petrinetzen gleiche Kantenanschriften auch immer für den Fluß gleicher Marken sorgen, ist dies in FSNets nur für Ausgangskanten der Fall. Transitionen in FSNets haben die Aufgabe, vorhandene Information zu unifizieren. Deswegen müssen über gleich beschriftete Eingangskanten fließende Marken nicht gleich sein, sondern nur unifizierbar. Sie werden sozusagen beim Schalten gleichgemacht. Dies entspricht genau der Idee der Feature Structure als Repräsentant partieller Information (siehe Abschnitt 2.3.1). Information, die in einer Feature Structure nicht explizit negativ gegeben wird (siehe dazu Kapitel 7 in [Carpenter 1992]), gilt als (noch) nicht bekannt.

Der Formalismus bietet so eine sehr kompakte Darstellung der Unifikation von Information analog zum Steuerfluß. So unifiziert eine Transition mit mehreren Eingangskanten und einer oder mehreren Ausgangsstellen alle Marken von den Eingangsstellen und legt deren Unifikation auf der oder den Ausgangsstelle(n) ab, ohne daß man eine einzige Kantenanschrift benötigt. Eine Falle hält diese Notation allerdings bereit: Solange man keine speziellen Reservierungs- oder Testkanten definiert (siehe Abschnitt 4.4.2), kann man Marken „aus Versehen“ ändern, die man eigentlich nur als Nebenbedingung verwenden wollte, die aber ihren abstrakteren Wert bewußt behalten sollen.

4.2 Elementare Feature-Structure-Netze

In diesem Abschnitt wird eine vereinfachte Version der in diesem Kapitel eingeführten Basis-FSNets definiert, die analog zu elementaren Netzsystem, also einem markierten B/E-Netz (siehe Abschnitt 3.2.1) *elementare Feature-Structure-Netze* (*elementary feature-structure-nets*, elementare FSNets oder EFSNets) genannt werden. Die Vereinfachungen haben zum Ziel, die Betrachtung weiterer Konzepte im Rahmen dieser Arbeit möglich zu machen. Zum einen wird die Diskussion von Wertversus Referenzsemantik wieder aufgegriffen, zum anderen werden für elementare FSNets erweiterte Ausführungssemantiken definiert: eine Schrittsemantik, die das gleichzeitige Schalten mehrerer Transitionen in einem Schritt erlaubt, und eine Prozeßsemantik, die echte Nebenläufigkeit von Transitionen repräsentieren kann. Dieses erlaubt wiederum, einen engen Bezug zu Valks Netzen in Netzen und Kummers Referenznetzen herstellen zu können. Die Einschränkungen wurden so gewählt, daß sie

immer noch einen praktischen Gebrauch von FSNets zulassen. Im Prinzip wäre es auch möglich, die folgenden Betrachtungen für Basis-FSNets durchzuführen, worauf aber aus Gründen der Übersichtlichkeit, Verständlichkeit und des Aufwands verzichtet wurde. Um die Überlegungen auf Basis-FSNets zu übertragen, wäre eine Erweiterung der Feature Structures auf mengen- oder gar multimengenwertige Features notwendig, die den Rahmen des formalen Teils dieser Arbeit sprengen würde. Eine Darstellung von Multimengen als Listen wäre zwar möglich, führte aber zu unbefriedigenden Ergebnissen in bezug auf Gleichheit von Strukturen und zu einem unerwünschten Unifikationsverhalten, da Listen stets eine bestimmte Reihenfolge der Elemente spezifizieren.

Im wesentlichen beschränken wir uns bei elementaren FSNets auf höchstens eine Feature-Structure-Marke pro Stelle und auf entsprechend vereinfachte Kantenanschriften und Markierungen und vermeiden somit mengenwertige Features. Wir verzichten weiterhin auch auf eine Typisierung der Stellen, da diese durch entsprechende Transitionsregeln simuliert werden kann. Bevor elementare FSNets formal definiert werden, wird im folgenden Abschnitt die Diskussion um Wert- und Referenzsemantik anhand eines Beispiels nochmals aufgegriffen.

4.2.1 Motivation: Wert- versus Referenzsemantik in elementaren FSNets

Die Vereinfachungen in elementaren FSNets führen unter anderem dazu, daß in Abschnitt 4.2.3 eine operationale Semantik von elementaren FSNet sowohl nach Wert- als auch nach Referenzsemantik definiert werden kann, womit die Diskussion aus Abschnitt 3.3.4 wieder aufgegriffen wird.

In einer Referenzsemantik für FSNets sollte es analog zu den Überlegungen aus Abschnitt 3.3.4 möglich sein, daß verschiedene Marken, in diesem Fall Feature Structures, einander referenzieren oder anders gesagt Strukturen teilen. In einer Wertsemantik wird dagegen wie bisher jede Marke als eigenständige Feature Structure betrachtet, so daß das Teilen von Strukturen zwischen verschiedenen Marken nicht möglich ist.

Bevor EFSNets formalisiert werden, soll der Unterschied zwischen Wert- und Referenzsemantik anhand eines Beispiels deutlich gemacht werden. Es wird gezeigt, welche inhaltlichen Implikationen aus der Anwendung welcher Semantik folgen. Dazu wird das Beispiel aus Abschnitt 4.1.2 wieder aufgegriffen.

Das dort gezeigte Netz diente zum Umkehren einer Liste und kann (bis auf die Stellentypisierungen) auch als elementares FSNet aufgefaßt werden, da zu einem Zeitpunkt nie mehr als eine Marke auf jeder Stelle liegt. Der Unterschied zwischen Wert- und Referenzsemantik kann an diesem Beispiel nicht gezeigt werden, da nur genau eine Feature-Structure-Marke zur Zeit besteht. Wir verwenden deswegen die in Abbildung 4.10 gezeigte modifizierte Version, die eine Liste aus einem anfangs unspezifizierten ersten Element und einer Feature Structure [B] umkehrt und ne-

benläufig ergänzt. Um die später dargestellten Schaltfolgen und Prozesse zu vereinfachen, besteht die Liste nun nur noch aus zwei Elementen. Außerdem soll als Ergebnis nicht nur die umgekehrte Liste, sondern eine Feature Structure vom Typ Result geliefert werden, die unter dem Feature forwards die vorwärtsnotierte und unter backwards die umgekehrte Liste enthält.

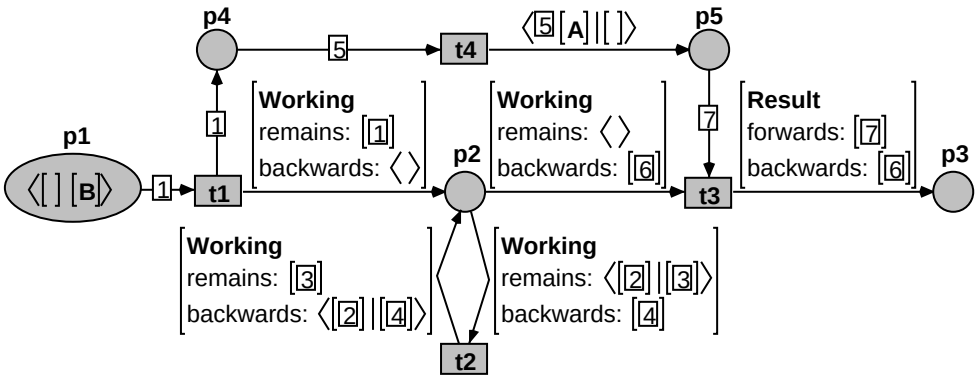

Abbildung 4.10: Ein elementares FSNet, das eine Liste umkehrt und nebenläufig ergänzt. Dieses Netz funktioniert nur sinnvoll mit einer Referenzsemantik.

Transition t1 übernimmt die Liste wie zuvor in die Marke, die auf die Stelle p2 gelegt wird. Zusätzlich wird nun aber die Liste auf p4 gelegt.[2] Durch t4 wird das erste Element der Liste konkretisiert, und die sich ergebende Liste wird auf p5 abgelegt.

t3 hat nun die Aufgabe, die umgekehrte und die bearbeitete Liste wieder zusammenzufügen. Hier stößt man auf den Unterschied zwischen Wert- und Referenzsemantik. Nehmen wir für die Ausführung des Netzes eine Wertsemantik an, so bleibt die Liste in der Marke in p2 von der Änderung durch t4 unberührt. Da die Liste in p2 aber auch bearbeitet (nämlich umgekehrt) wird, stellt es sich als schwierig heraus, die durch t4 hinzugefügte Information („Das erste Element der Liste hat den Typ A.“) nachzutragen. In diesem Fall führt eine Referenzsemantik zum gewünschten Verhalten. Die in p4 abgelegte Liste ist dann in der Tat *dieselbe* wie die in der Marke in p2 unter dem Feature remains enthaltene. Jede Änderung an einem Knoten der Feature Structure in p4 ändert den entsprechenden Knoten der Marke in p2, da diese dieselben Knoten sind.

Man kann entsprechendes Verhalten in der bisher benutzten Wertsemantik nachahmen. Für das nachträgliche Wiedervereinigen von Information eignet sich die Unifikation von Feature Structures besonders gut. Abbildung 4.11 zeigt eine Lösung, die mit einer Wertsemantik das gewünschte Ergebnis produziert. Um zu wissen, welche

[2] Ob es sich hierbei um dieselbe Liste oder um eine Kopie handelt, ist gerade Inhalt der folgenden Diskussion um Wert- versus Referenzsemantik.

Knoten der bearbeiteten Feature Structures unifiziert werden müssen, ist es in diesem Beispiel nötig, während des Umkehrens eine Kopie der Liste in derselben Marke aufzubewahren. Dazu dient das Feature forwards im Typ Working2, der die Features von Working erbt. Sodann kann die im nebenläufigen Zweig bearbeitete Liste durch t3 mit der Kopie der Originalliste unifiziert werden, was die neu hinzugekommene Information auch in die umgekehrte Liste propagiert.

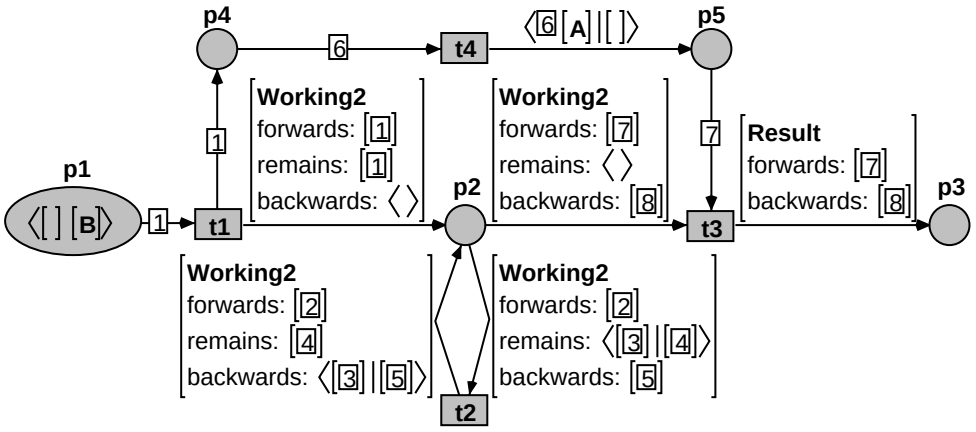

Abbildung 4.11: Ein elementares FSNet, das unter Verwendung von Wertsemantik das Verhalten des Netzes aus Abbildung 4.10 nachbildet.

4.2.2 Formale Definition von elementaren FSNets

In diesem Abschnitt werden die im vorherigen Abschnitt informal eingeführten EFSNets definiert. Das wesentliche Merkmal von EFSNets im Vergleich mit Basis-FSNets ist, daß diese höchstens eine Marke pro Stelle erlauben. Es handelt sich also um eine Art gefärbtes B/E-Netz.

Es wäre einfach, EFSNets als eingeschränkte Basis-FSNets zu definieren, doch würde dies nicht zu einem einfacher zu handhabenden Modell führen, sondern es im Gegenteil nötig machen, die Einschränkungen zusätzlich zu behandeln. Bei der Definition von Basis-FSNets in Abschnitt 4.1.1 mußten für mehrere Konstrukte totale Funktionen nach Multimengen von Pfaden bzw. Feature Structures eingesetzt werden. Durch die Beschränkung der EFSNets können diese durch partielle Funktionen nach Pfaden bzw. Feature Structures ersetzt werden.

Weiterhin sollen EFSNets als eigenständiger Formalismus betrachtet werden, da für diese im Gegensatz zu Basis-FSNets zusätzlich eine *Referenzsemantik* betrachtet werden soll. Nur nach Wertsemantik verhalten sich EFSNets wie eine Einschränkung von Basis-FSNets.

Wenden wir uns zunächst der Realisierung einer Referenzsemantik für elementare FSNets zu. Innerhalb einer Feature Structure ist es durch Knotenreferenzen kein Problem, Referenzen auszudrücken. Konsequenterweise wird eine Markierung eines elementaren FSNet nicht mehr als Funktion von Stellen zu Feature Structures mit disjunkten Knotenmengen beschrieben, sondern durch eine einzige Feature Structure, welche die Markierung des gesamten Netzes enthält. Dafür werden für jede Stelle ein Feature sowie zwei Typen M und E eingeführt, wobei M für eine Markierung steht und alle Stellen-Features mit dem Typ $\top$ erlaubt. Unmarkierte Stellen werden in elementaren FSNets über eine Marke vom Typ E dargestellt, um die Kapazität Eins für alle Stellen sicherzustellen, ohne Komplementärstellen konstruieren zu müssen (siehe Abschnitt 3.2.2). Da es sich bei elementaren FSNets um gefärbte Netze handelt, kann die Funktion der Komplementärstelle durch diese spezielle Marke simuliert werden.

Definition 4.6 [Markierungs-Typsystem]
Sei $\mathrm{TS} = \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ ein Typsystem, $\top$ der allgemeinste Typ aus TS und S eine Menge von Stellen. Das *Markierungs-Typsystem* $\mathrm{MTS}_{\mathrm{TS,S}}$ zu TS und S ergibt sich dann als $\mathrm{MTS}_{\mathrm{TS,S}} := \langle \mathrm{Type} \cup \mathrm{Type}_{\mathrm{M}}, (\sqsubseteq_{\mathrm{M}}), \mathrm{Feat} \cup \mathrm{S}, (\mathrm{approp}) \cup (\mathrm{approp}_{\mathrm{M}}) \rangle$ mit:

(a) $\mathrm{Type}_{\mathrm{M}} := \{\top_{\mathrm{M}}, \mathsf{M}, \mathsf{Token}, \mathsf{E}\}$,

(b) $(\sqsubseteq_M)$ ist die reflexive, transitive Hülle über $(\sqsubseteq) \cup (\{\top_{\mathrm{M}}\} \times \{\mathsf{M}, \mathsf{Token}\}) \cup (\{\mathsf{Token}\} \times \{\mathsf{E}, \top)\})$,

(c) $\forall s \in \mathrm{S} : \mathrm{approp}_{\mathrm{M}}(\mathsf{M}, s) := \top$, ansonsten sei $\mathrm{approp}_{\mathrm{M}}$ undefiniert.

O.B.d.A. sei $\mathrm{Type} \cap \mathrm{Type}_{\mathrm{M}} = \emptyset$ und $\mathrm{Feat} \cap \mathrm{S} = \emptyset$. ◊

Führt man diese Konstruktion für das Typsystem aus Abbildung 4.1 und die Stellenmenge $\mathrm{S} = \{\mathrm{p1}, \mathrm{p2}, \mathrm{p3}, \mathrm{p4}, \mathrm{p5}\}$ durch, so erhält man das in Abbildung 4.12 dargestellte Typsystem. Man beachte, daß M für jede Stelle ein Feature $\mathsf{p1}$, $\mathsf{p2}$, ..., $\mathsf{p5}$ erlaubt.

Bevor elementare FSNets definiert werden, sei zunächst eine Markierung definiert, da auf diese für die Definition der Anfangsmarkierung zurückgegriffen werden soll. Eine solche einfache Markierung kann nur eine oder keine Marke pro Stelle repräsentieren. Dafür läßt sie sich wie oben bereits erwähnt als *eine* Feature Structure darstellen, die für jede Stelle ein Feature definiert. Leere Stellen werden durch einen Knoten vom Typ E repräsentiert.

Definition 4.7 [Elementarmarkierung]
Eine *Elementarmarkierung* über ein Typsystem TS und eine Menge von Stellen S ist eine Feature Structure $\mathrm{m} \in \mathrm{WFS}_{\mathrm{MTS}_{\mathrm{TS,S}}}$ mit $\theta(\mathrm{m}) = \mathsf{M}$, wobei $\mathrm{MTS}_{\mathrm{TS,S}}$ das Markierungs-Typsystem zu TS und S ist. Für m muß jedes in M erlaubte Feature auch definiert sein, also es muß gelten: $\forall s \in \mathrm{S} : \mathrm{m}@s$ ist definiert. Eine Stelle $s \in \mathrm{S}$ heißt *leer* oder *unmarkiert*, wenn gilt $\mathrm{m}@s \sim [\mathsf{E}]$, ansonsten heißt die Stelle *markiert mit* $\mathrm{m}@s$. ◊

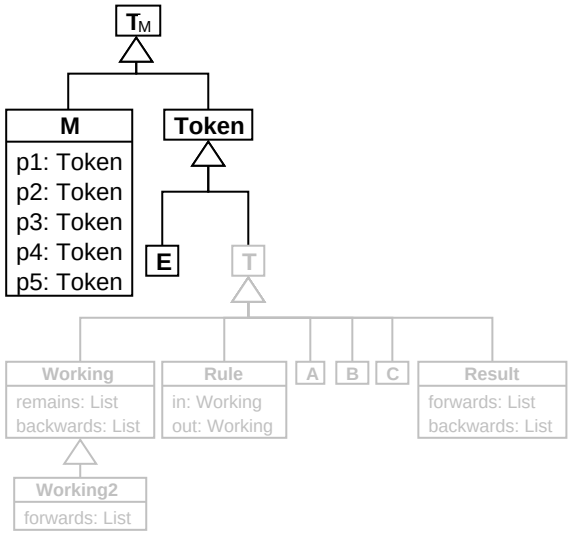

Abbildung 4.12: Das Markierungs-Typsystem zum Typsystem aus Abbildung 4.1 und zur Stellenmenge $S = \{p1, p2, p3, p4, p5\}$.

Nun läßt sich ein elementares FSNet wie folgt definieren.

Definition 4.8 [Elementares FSNet]
Ein elementares Feature-Structure-Netz (elementares FSNet) (EFSNet) ist ein Tupel $EFSN = \langle S, T, TS, l, r, m_0 \rangle$ aus folgenden Komponenten:

(a) einer endlichen Menge von *Stellen* S und einer endlichen Menge von *Transitionen* T mit $S \cap T = \emptyset$,

(b) einem *Typsystem* $TS = \langle Type, \sqsubseteq, Feat, approp \rangle$ (Def. 2.4),

(c) einer partiellen *Kantenanschriftfunktion* $l : (S \times T) \cup (T \times S) \hookrightarrow Feat^*$, die jeder existierenden Kante einen Pfad zuordnet, wobei sich die Flußrelation aus der Def. von Petrinetzen als $F = dom(l)$ ergibt und es keine Transitionen ohne Wirkung geben darf: $\forall t \in T : \exists s \in S : l(s,t)$ ist definiert $\vee$ $l(t,s)$ ist definiert,

(d) einer *Transitionsregelfunktion* $r : T \rightarrow WFS_{TS}$, die jeder Transition eine wohltypisierte Feature Structure als Schaltregel zuweist und

(e) einer Anfangsmarkierung m_0, die eine Elementarmarkierung über TS und S darstellt.

Damit ein elementares FSNet statisch wohlgeformt ist, muß gelten:

$$\begin{aligned} \forall t \in T, s \in S : \quad & (l(s,t) \text{ ist definiert } \Rightarrow r(t)@l(s,t) \text{ ist definiert}) \\ \wedge \quad & (l(t,s) \text{ ist definiert } \Rightarrow r(t)@l(t,s) \text{ ist definiert}) \end{aligned}$$

(die Transitionsregeln enthalten die Pfade aus den entsprechenden Kantenanschriften). ◊

In der Definition werden Transitionen ohne Wirkung ausgeschlossen, da diese in Petrinetzen allgemein zu unerwünschten Eigenschaften führen (siehe [Baumgarten 1990]) und gerade bei der Konstruktion von Prozessen als Sonderfall behandelt werden müßten. Transitionen ohne Wirkung sind Transitionen ohne Vor- *und* Nachbedingungen, also solche, die mit keiner Stelle verbunden sind. Diese müßten nach einer echten Petrinetzsemantik beliebig oft nebenläufig zu sich selbst schalten können. Da ihr Schalten aber keinerlei Einfluß auf das restliche System hat, sollten solche Transitionen ohnehin aus Sicht der Modellierung vermieden werden.

Wie oben bereits gesagt schalten Basis-FSNets immer nach Wertsemantik, wogegen für elementare FSNets im nächsten Abschnitt sowohl eine Wert- als auch eine Referenzsemantik definiert wird. Für den Fall der Wertsemantik kann man ein EFSNet $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ auf ein Basis-FSNet $BFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{s}_{\mathrm{type}}, \mathrm{l}', \mathrm{r}, \mathrm{m}_0' \rangle$ mit äquivalentem Schaltverhalten abbilden, wenn für das Basis-FSNet zugesichert wird, daß jede Stelle die Kapazität 1 besitzt.

Man bildet ein EFSNet $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ auf ein Basis-FSNet $BFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{s}_{\mathrm{type}}, \mathrm{l}', \mathrm{r}, \mathrm{m}_0' \rangle$ ab, indem man die fehlenden oder komplexeren Komponenten wie folgt rekonstruiert. $\mathrm{s}_{\mathrm{type}}$ wird so definiert, daß jede Stelle den Typ $[\top]$ erhält. $\mathrm{l}'(\mathrm{x}, \mathrm{y})$ wird bestimmt, indem jeder Kante (x, y), für die $\mathrm{l}(x, y)$ definiert ist, die einelementige Multimenge $\{\mathrm{l}(x, y)\}_{\mathrm{M}}$ zugewiesen wird, die leere Multimenge sonst. Die 1-Sicherheit kann leicht durch Konstruktion entsprechender Komplementärstellen erreicht werden (siehe Abschnitt 3.2.2). Die Anfangsmarkierung m_0' wird rekonstruiert als

$$\forall s \in \mathrm{S} : \mathrm{m}_0'(s) := \begin{cases} \emptyset_{\mathrm{M}} & \text{falls } \mathrm{m}_0@s \sim [\mathsf{E}] \\ \{\mathrm{m}_0@s\}_{\mathrm{M}} & \text{sonst} \end{cases}$$

Damit sind alle Komponenten des EFSNet auf ein Basis-FSNet abgebildet.

4.2.3 Schaltfolgen in elementaren FSNets nach Wert- und Referenzsemantik

Im vorherigen Abschnitt wurde eine Abbildung von EFSNets auf Basis-FSNets skizziert. Mit Hilfe einer solchen Abbildung könnte die Definitionen von Aktivierung und Schalten in Basis-FSNets für elementare FSNets übernommen werden. Die Definition soll aber speziell für elementare FSNets umformuliert und vereinfacht werden, da so Wert- und Referenzsemantik durch einfache Fallunterscheidungen definiert und sogar innerhalb eines Netzes verwendet werden können (siehe Abschnitt 4.2.5).

Anhand einer Elementarmarkierung läßt sich definieren, wie sich Wert- und Referenzsemantik unterscheiden.

Definition 4.9 [Wertmarkierung]
Sei $\mathrm{m_v}$ eine Elementarmarkierung über einem Typsystem TS $= \langle \mathrm{Type}, \sqsubseteq, \mathrm{Feat}, \mathrm{approp} \rangle$ und einer Stellenmenge S und $\mathrm{MTS_{TS,S}}$ das Markierungs-Typsystem zu TS und S. $\mathrm{m_v} \in \mathrm{WFS_{MTS_{TS,S}}}$ heißt eine *Wertmarkierung* genau dann wenn gilt:

$$\forall s, s' \in \mathrm{S}, \pi, \pi' \in \mathrm{Feat}^* : \mathrm{m_v}@(s\,\pi) = \mathrm{m_v}@(s'\,\pi') \Rightarrow s = s'$$

(zeigen zwei Pfade auf dieselbe Unterstruktur, so müssen sie mit demselben Feature beginnen, also muß jede Unterstruktur in genau einer Marke liegen). ◊

Eine Elementarmarkierung eines EFSNet darf also beliebige Strukturgleichheiten zwischen verschiedenen Marken enthalten, während eine *Wert*markierung nur Strukturgleichheiten innerhalb einer Marke erlaubt.

Man kann aus einer beliebigen Elementarmarkierung eine Wertmarkierung generieren, indem man geteilte Unterstrukturen kopiert.

Definition 4.10 [Konstruktion einer Wertmarkierung]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m_0} \rangle$ ein elementares FSNet und $\mathrm{MTS_{TS,S}}$ das zugehörige Markierungs-Typsystem. Der Wertmarkierungskonstruktor valmark : $WFS_{MTS_{TS,S}} \hookrightarrow WFS_{MTS_{TS,S}}$ ist eine Funktion, die für eine Elementarmarkierung $\mathrm{m} \in \mathrm{WFS_{MTS_{TS,S}}}$ folgendermaßen definiert sei:

$$\mathrm{valmark(m)} := [\mathsf{M}] \sqcup_{\mathrm{U}} \bigsqcup_{\mathrm{U}} \{s::(\mathrm{m}@s) \mid s \in \mathrm{S}\} \qquad \Diamond$$

Das Ergebnis valmark(m) ist trotz Anwendung der untypisierten Unifikation in der Konstruktion aus den folgenden Gründen wohltypisiert. Die Elementarmarkierung m ist wohltypisiert, und damit offensichtlich auch jede Substruktur von m. valmark(m) könnte also nur dann nicht-wohltypisiert sein, wenn der Wurzelknoten unerlaubte Features enthielte oder diese auf Knoten eines nicht angemessenen Typs zeigten. Der Typ M ist aber für alle Features $s \in \mathrm{S}$ definiert, für die in der Konstruktion vom Wurzelknoten abgehende Kanten erzeugt werden, und erlaubt für die Werte dieser Features beliebige Typen ($\top$), so daß valmark(m) wohltypisiert ist.

Durch diese Konstruktion erhält man tatsächlich die speziellste mögliche Wertmarkierung, welche die Elementarmarkierung subsumiert, wie der folgende Satz aussagt. Dies ist genau die Eigenschaft, die für die Konstruktion einer Wertmarkierung erwünscht ist, da so sichergestellt ist, daß nur die Koreferenzen zwischen unterschiedlichen Marken aufgehoben werden, ansonsten aber keine weitere Information entfernt wird.

Dieser Satz ist somit eine zentrales Ergebnis dieser Arbeit zu der Fragestellung, wie sich Wert- und Referenzsemantik zueinander verhalten. Es wird gezeigt, daß zu jeder Referenzmarkierung eine (bis auf alphabetische Varianten) eindeutige Wertmarkierung konstruiert werden kann, welche bis auf die unerlaubten Koreferenzen die gleiche Information enthält wie die Referenzmarkierung.

Satz 4.11 [valmark **erzeugt speziellste Wertmarkierung**]
Sei $TS = \langle Type, \sqsubseteq, Feat, approp\rangle$ ein Typsystem, S eine Stellenmenge und $MTS_{TS,S}$ das Markierungs-Typsystem zu TS und S. Die in Definition 4.10 konstruierte Funktion valmark liefert zu einer Elementarmarkierung $m \in WFS_{MTS_{TS,S}}$ die speziellste Wertmarkierung valmark(m), die m subsumiert.

Beweis Sei $m \in WFS_{MTS_{TS,S}}$ eine Elementarmarkierung und $m_v := valmark(m)$. Es ist zu zeigen, daß m_v eine Wertmarkierung ist und daß für alle Wertmarkierungen $m' \in WFS_{MTS_{TS,S}}$ gilt: $m' \sqsubseteq m \Rightarrow m' \sqsubseteq m_v$.
Wir zeigen zuerst, daß m_v eine Elementarmarkierung ist. Dafür müssen wir zeigen, das $\theta(m_v) = \mathsf{M}$ und daß für jedes $s \in S$ gilt, daß $m_v@s$ definiert ist.
Nach Konstruktion ist

$$\begin{aligned}\theta(m_v) &= \theta([\mathsf{M}] \sqcup_U \bigsqcup_U \{s{::}(m@s) \mid s \in S\}) \\ &= \mathsf{M} \sqcup \theta(\bigsqcup_U \{s{::}m@s \mid s \in S\})) \\ &= \mathsf{M} \sqcup \top \\ &= \mathsf{M}.\end{aligned}$$

Für die Pfadwerte für alle Stellen ergibt sich:

$$\begin{aligned}\forall s \in S : m_v@s &= ([\mathsf{M}] \sqcup_U \bigsqcup_U \{s'{::}(m@s') \mid s' \in S\})@s \\ &= ([\mathsf{M}] \sqcup_U \bigsqcup_U \{s'{::}(m@s') \mid s' \in S\})@s \\ &= ([\mathsf{M}] \sqcup_U \bigsqcup_U \{s'{::}(m@s') \mid s' \in S - \{s\}\} \sqcup_U s{::}(m@s))@s \\ &\sim (s{::}(m@s))@s && \text{(nach Satz 2.36)} \\ &= m@s && \text{(nach Satz 2.27)}\end{aligned}$$

Da für m als Elementarmarkierung nach Definition 4.7 gilt, daß $m@s$ für alle Stellen-Features s definiert ist, folgt aus $m_v@s \sim m@s$, daß auch für jedes $s \in S$ gilt: $m_v@s$ ist definiert.
Es muß noch gezeigt werden, daß m_v die Wertmarkierungs-Eigenschaft besitzt, also daß

$$\forall s, s' \in S, \pi, \pi' \in Feat^* : m_v@(s\,\pi) = m_v@(s'\,\pi') \Rightarrow s = s' \tag{4.3}$$

Nehmen wir an, daß über zwei Pfade $s\,\pi$ und $s'\,\pi'$ mit $s \neq s'$ derselbe Knoten erreicht wird. In der Konstruktion von valmark(m) werden vor der Unifikation der unter Vorpfade gestellten Pfadwerte (wie bei jeder Unifikation) die Knoten auf disjunkte Knotenmengen umbenannt. Wenn im Unifikationsergebnis über zwei unterschiedliche Pfade derselbe Knoten erreicht wird, dann nur, weil dies durch eine an der Unifikation beteiligten Feature Structure gegeben ist, oder weil die Knoten während der Unifikation identifiziert werden. Der erste Fall kann ausgeschlossen werden, da in jeder an der Unifikation beteiligten Feature Structure alle definierten Pfade mit einer Länge größer 1 mit demselben Feature s beginnen. Bei der Unifikation in der Konstruktion von valmark(m) ergibt sich dadurch weiterhin wie in Satz 2.36 die triviale Äquivalenzrelation, die lediglich alle Wurzelknoten identifiziert. Deshalb kann

auch der zweite Fall ausgeschlossen werden, daß weitere Knoten identifiziert werden, womit die Behauptung widerlegt ist.
Weiterhin ist zu zeigen, daß $\mathrm{m_v}$ die speziellste Wertmarkierung ist, die m subsumiert. Wir zeigen zunächst, daß $\mathrm{m_v} \sqsubseteq \mathrm{m}$. Da m als Elementarmarkierung für alle Stellen-Features definiert sein muß, gilt nach Satz 2.27:

$$\forall s \in \mathrm{S} : s{::}(\mathrm{m}@s) \sqsubseteq \mathrm{m}$$

Wegen $\theta(\mathrm{m}) = \mathsf{M}$ gilt weiterhin $[\mathsf{M}] \sqsubseteq \mathrm{m}$. Nach den Sätzen 2.33 und 2.34 und den Eigenschaften der Unifikation folgt aus $F \sqsubseteq \hat{F}$ und $F' \sqsubseteq \hat{F}$, daß $F \sqcup_\mathrm{U} F' \sqsubseteq \hat{F}$, für den konkreten Fall also

$$\begin{aligned} [\mathsf{M}] \sqcup_\mathrm{U} \bigsqcup_\mathrm{U} \{s{::}(\mathrm{m}@s) \mid s \in \mathrm{S}\} &\sqsubseteq \mathrm{m} \\ \iff \mathrm{m_v} &\sqsubseteq \mathrm{m} \end{aligned}$$

Es bleibt zu zeigen, daß für alle Wertmarkierungen $\mathrm{m}' = \langle \mathrm{Q}', \bar{\mathrm{q}}', \theta', \delta' \rangle$ gilt:

$$\mathrm{m}' \sqsubseteq \mathrm{m} \Rightarrow \mathrm{m}' \sqsubseteq \mathrm{m_v} \tag{4.4}$$

Sei $\langle \mathrm{Q_v}, \bar{q}_\mathrm{v}, \theta_\mathrm{v}, \delta_\mathrm{v} \rangle := \mathrm{m_v}$. Da $\mathrm{m_v}$ und m' Elementarmarkierungen sind, sind die folgenden Substrukturen definiert: Seien $\forall s \in \mathrm{S} : \langle \mathrm{Q}_s, \bar{q}_s, \theta_s, \delta_s \rangle := \mathrm{m_v}@s$ und $\langle \mathrm{Q}'_s, \bar{q}'_s, \theta'_s, \delta'_s \rangle := \mathrm{m}'@s$. Da $\mathrm{m_v}$ und m' weiterhin Wertmarkierungen sind, ergibt sich aus Definition 4.9, daß für alle $s, s' \in \mathrm{S}$ gilt:

$$s \neq s' \Rightarrow (\mathrm{Q}_s \cap \mathrm{Q}_{s'} = \emptyset \wedge \mathrm{Q'}_s \cap \mathrm{Q'}_{s'} = \emptyset). \tag{4.5}$$

Nach Aussage 4.4 und Satz 2.30 folgt, daß für alle $s \in \mathrm{S}$ gilt $\mathrm{m}'@s \sqsubseteq \mathrm{m}@s$. Zusammen mit der oben gezeigten Aussage $\mathrm{m_v}@s \sim \mathrm{m}@s$ folgt, daß für alle $s \in \mathrm{S}$ gilt $\mathrm{m}'@s \sqsubseteq \mathrm{m_v}@s$. Es gibt also für jedes $s \in \mathrm{S}$ eine Subsumptions-Funktionen $\mathrm{h}_s : \mathrm{Q}'_s \to \mathrm{Q}_s$.
Zu zeigen ist, daß es eine Subsumptions-Funktion $\mathrm{h} : \mathrm{Q}' \to \mathrm{Q_v}$ gibt. Diese Funktion sei folgendermaßen definiert: $\mathrm{h}(\bar{q}') := \bar{q}_\mathrm{v}$ und $\forall s \in \mathrm{S}, q \in \mathrm{Q}'_s : \mathrm{h}(q) := \mathrm{h}_s(q)$. Damit ist h eindeutig und vollständig definiert, da die Q_s paarweise disjunkt sind (Aussage 4.5) und eine Elementarmarkierung als wohltypisierte Feature Structure keine weiteren als die Stellen-Features enthalten darf und zusammenhängend ist, also gilt $\mathrm{Q}' = \{\bar{q}'\} \cup \bigcup \{\mathrm{Q}'_s \mid s \in \mathrm{S}\}$. Weiterhin gilt nach Konstruktion, daß $\mathrm{Q}_s \subseteq \mathrm{Q_v}$, also ist auch der Bildbereich von h wohldefiniert.
Nun ist zu zeigen, daß h die in Definition 2.29 genannten Eigenschaften (a) bis (c) erfüllt.
Der Wurzelknoten bleibt nach Konstruktion von h trivialerweise erhalten, da $\mathrm{h}(\bar{q}') = \bar{q}_\mathrm{v}$.

Die Forderung nach mindestens so speziellen Typen ist für den Wurzelknoten erfüllt, da $\theta'(\bar{q}') = \theta_{\mathrm{v}}(\bar{q}_{\mathrm{v}}) = \mathsf{M}$. Für alle anderen Knoten $q \in \mathrm{Q}' - \{\bar{q}'\}$ gibt es ein $s \in \mathrm{S}$, so daß $q \in \mathrm{Q}'_s$. Für die Subsumptionsfunktion h_s gilt:

$$\begin{aligned} \theta'_s(q) &\sqsubseteq \theta_s(\mathrm{h_s}(q)) \\ \Rightarrow \theta'(q) &\sqsubseteq \theta_{\mathrm{v}}(\mathrm{h}(q)). \end{aligned}$$

Für die Abbildung der Kanten ist eine Fallunterscheidung zu treffen. Für $\delta'(q_1, f) = q_2$ ist entweder $q_1 = \bar{q}'$ oder beide Knoten müssen in derselben Feature Structure $\mathrm{m}'@s$ liegen ($\exists s \in \mathrm{S} : q_1, q_2 \in \mathrm{Q}'_s$), da m' eine Wertmarkierung ist. Für den ersten Fall gilt:

$$\delta'(\bar{q}', s) = q_2 = \bar{q}'_s \implies \delta_{\mathrm{v}}(\mathrm{h}(\bar{q}'), s) = \delta_{\mathrm{v}}(\bar{q}_{\mathrm{v}}, s) = \bar{q}_s = \mathrm{h}_s(\bar{q}'_s) = \mathrm{h}(\bar{q}'_s) = \mathrm{h}(q_2).$$

Für den zweiten Fall ergibt sich

$$\begin{aligned} \delta'(q_1, f) &= q_2 \\ \Rightarrow \delta'_s(q_1, f) &= q_2 \\ \Rightarrow \delta_s(\mathrm{h}_s(q_1), f) &= \mathrm{h}_s(q_2) \\ \Rightarrow \delta_{\mathrm{v}}(\mathrm{h}(q_1), f) &= \mathrm{h}(q_2). \end{aligned}$$

Damit ist gezeigt, daß jede andere Wertmarkierung m', welche die Elementarmarkierung m subsumiert, auch die konstruierte Wertmarkierung valmark(m) subsumiert, und damit, daß valmark(m) die speziellste Wertmarkierung darstellt, welche die Elementarmarkierung m subsumiert. □

Aus den Kanten- und Transitionsanschriften läßt sich eine abgeleitete Transitionsregel herleiten, die sich zur Definition von Aktivierung und Schalten von Transitionen in elementaren FSNets eignet. Dafür wird wie zuvor ein Hilfs-Typsystem konstruiert, in dem die abgeleiteten Transitionsregeln als wohltypisierte Feature Structures dargestellt werden können.

Definition 4.12 [Transitionsregel-Typsystem]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet, $\mathrm{MTS}_{\mathrm{TS,S}} = \langle \mathrm{Type}_{\mathrm{M}}, \sqsubseteq_{\mathrm{M}}, \mathrm{Feat}_{\mathrm{M}}, \mathrm{approp}_{\mathrm{M}} \rangle$ das Markierungs-Typsystem zu TS und S, $\top \in \mathrm{TS}$ der allgemeinste Typ von TS und $\top_{\mathrm{M}} \in \mathrm{MTS}_{\mathrm{TS,S}}$ der allgemeinste Typ des Markierungs-Typsystems. Dann sei das *Transitionsregel-Typsystem* $\mathrm{RTS}_{EFSN} := \langle \mathrm{Type}_{\mathrm{M}} \cup \mathrm{Type}_{\mathrm{R}}, (\sqsubseteq_{\mathrm{M}}) \cup (\sqsubseteq_{\mathrm{R}}), \mathrm{Feat}_{\mathrm{M}} \cup \mathrm{Feat}_{\mathrm{R}}, (\mathrm{approp}_{\mathrm{M}}) \cup (\mathrm{approp}_{\mathrm{R}}) \rangle$ mit

(a) $\mathrm{Type}_{\mathrm{R}} := \{\mathsf{Tr}, \mathsf{Eff}\}$,

(b) $(\sqsubseteq_{\mathrm{R}})$ sei die reflexive Hülle über $\{\top_M\} \times \{\mathsf{Tr}, \mathsf{Eff}\}$,

(c) $\mathrm{Feat}_{\mathrm{R}} := \{\mathsf{rule}, \mathsf{eff}, \mathsf{pre}, \mathsf{post}\}$,

(d) $\mathrm{approp_r}(\mathsf{Tr}, \mathsf{rule}) := \top$, $\mathrm{approp_r}(\mathsf{Tr}, \mathsf{eff}) := \mathsf{Eff}$, $\mathrm{approp_r}(\mathsf{Eff}, \mathsf{pre}) := \mathsf{M}$ und $\mathrm{approp_r}(\mathsf{Eff}, \mathsf{post}) := \mathsf{M}$,
ansonsten sei $\mathrm{approp_R}$ undefiniert.

O.B.d.A. sei $\mathrm{Type_M} \cap \mathrm{Type_R} = \mathrm{Feat_M} \cap \mathrm{Feat_R} = \emptyset$. ◊

Mit Hilfe dieses Typsystems werden nun die abgeleiteten Transitionsregeln konstruiert.

Definition 4.13 [Abgeleitete Transitionsregelfunktion]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m_0} \rangle$ ein elementares FSNet und RTS_{EFSN} das Transitionsregel-Typsystem zu $EFSN$. Eine *abgeleitete Transitionsregelfunktion* $\mathrm{r_a} : \mathrm{T} \to \mathrm{WFS}_{\mathrm{RTS}_{EFSN}}$, die Transitionsregel und Kantenanschrift zusammenfaßt, werde folgendermaßen konstruiert.
Sei für alle $t \in \mathrm{T}$:

$$
\begin{aligned}
\mathrm{r_a}(t) := \bigsqcup \Bigg\{ & \begin{bmatrix} \mathsf{Tr} \\ \mathsf{eff}\colon \begin{bmatrix} \mathsf{Eff} \\ \mathsf{pre}\colon [\mathsf{M}] \\ \mathsf{post}\colon [\mathsf{M}] \end{bmatrix} \end{bmatrix}, && \text{(a)} \\
& \mathsf{rule}{::}\mathrm{r}(t), && \text{(b)} \\
& \bigsqcup_{\mathrm{U}} \{ \mathsf{eff\ pre}\, s{::}[\mathsf{E}] \mid s \in t\bullet - \bullet t \}, && \text{(c)} \\
& \bigsqcup_{\mathrm{U}} \{ \mathsf{eff\ post}\, s{::}[\mathsf{E}] \mid s \in \bullet t - t\bullet \}, && \text{(d)} \\
& \bigsqcup_{\mathrm{U}} \{ \mathsf{eff\ pre}\, s \doteq \mathsf{rule}\, \mathrm{l}(s,t) \mid s \in \bullet t \}, && \text{(e)} \\
& \bigsqcup_{\mathrm{U}} \{ \mathsf{eff\ post}\, s \doteq \mathsf{rule}\, \mathrm{l}(t,s) \mid s \in t\bullet \} \Bigg\} && \text{(f)} \\
& @\mathsf{eff} && \text{(g)}
\end{aligned}
$$

◊

Im folgenden werden die Formelzeilen im einzelnen motiviert.

(a) Hier wird die Grundstruktur der abgeleiteten Transitionsregel erzeugt, die durch Unifikation mit den folgenden Termen ergänzt wird.

(b) Die Transitionsregel wird unter das Feature rule gestellt, um eine Vermischung mit der restlichen Konstruktion zu verhindern.

(c) Diese durch Unifikation erzeugte Feature Structure stellt sicher, daß Ausgangsstellen, die keine Nebenbedingungen sind, vor dem Schalten leer sind.

(d) Dieser Term legt fest, daß Eingangsstellen, die keine Nebenbedingungen sind, nach dem Schalten leer sind.

(e) Dieser Term verbindet die Feature Structures aus dem Vorbereich mit den entsprechenden Knoten aus der Transitionsregel.

(f) Der letzte Term verbindet die Feature Structures aus dem Nachbereich mit den entsprechenden Knoten aus der Transitionsregel.

(g) Es wird der Pfadwert der Unifikation unter eff als abgeleitete Transitionsregel zugewiesen.

Alle aus den Marken nicht erreichbaren Knoten der Transitionsregel fallen bei der Bildung des Pfadwerts weg. Diese können keinen Einfluß auf den Effekt der Transition haben.

Um die Veränderungen, die das Schalten einer Transition in elementaren FSNets an der Markierung hervorruft, die sogenannte Wirkung, vollständig durch Feature Structures beschreiben zu können, steht noch die Behandlung von Marken aus, die durch die Transition unberührt bleiben. Dieser Punkt wird hier aus zwei Gründen getrennt behandelt: Erstens gehört der Erhalt der Marken, die sich weder im Vornoch im Nachbereich der Transition befinden, nicht zur eigentlichen Wirkung der Transition[3]. Zweitens kann durch diese Trennung die abgeleitete Transitionsregelfunktion bei der Definition der Schrittsemantik (Abschnitt 4.2.6) und der Prozeßsemantik für elementare FSNets (Abschnitt 4.2.8) wiederverwendet werden. Eine Behandlung der unberührten Marken durch jede einzelne Transition eignet sich nur für eine nicht-nebenläufige Semantik wie die Schaltfolgensemantik.

Es müssen nun entsprechende Pfadgleichungen konstruiert werden, die eine Teilmenge von Marken von der aktuellen Markierung in die Folgemarkierung kopieren.

Definition 4.14 [Kopieren unberührter Marken]
Sei S eine Menge von Stellen. Die Funktion copymark : $\mathcal{P}(\mathrm{S}) \to \mathrm{FS}_{\mathrm{TS_r}}$ erzeugt eine Feature Structure, die für die Teilmenge von gegebenen Stellen Pfadgleichungen erzeugt, die Marken aus einer Markierung pre in eine Folgemarkierung post kopieren. Es sei

$$\forall \mathrm{S_u} \subseteq \mathrm{S} : \mathrm{copymark}(\mathrm{S_u}) := \bigsqcup_{\mathrm{U}} \{\mathsf{pre}\, s \doteq \mathsf{post}\, s \mid s \in \mathrm{S_u}\}. \qquad \Diamond$$

Die gesamte Wirkung einer Transition ergibt sich damit als Unifikation der abgeleiteten Transitionsregel und den Kopien der von dieser Transition unberührten Marken.

Definition 4.15 [Wirkung einer Transition in elementaren FSNets]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m_0} \rangle$ ein elementares FSNet und RTS_{EFSN} das Transitionsregel-Typsystem zu $EFSN$. Die EFSNet-Wirkung einer Transition $t \in \mathrm{T}$ sei

[3] Vergleiche dazu das sogenannte *frame problem* und die *closed world assumption* in der Logik ([Shanahan 1997]).

definiert als eine Funktion $\mathrm{w} : \mathrm{T} \to \mathrm{WFS}_{\mathrm{RTS}_{EFSN}}$ mit

$$\mathrm{w}(t) := \mathrm{r_a}(t) \sqcup \mathrm{copymark}(\mathrm{S} - \bullet t - t\bullet). \qquad \Diamond$$

Wenn aus dem Kontext klar ist, daß eine Transition eines EFSNets und nicht etwa eine S/T-Netz-Transition gemeint ist, sprechen wir im folgenden statt von der EFSNet-Wirkung auch einfach von der Wirkung einer Transition. Die Konstruktion der Wirkung einer Transition sei anhand des Beispiels aus Abbildung 4.10 veranschaulicht.

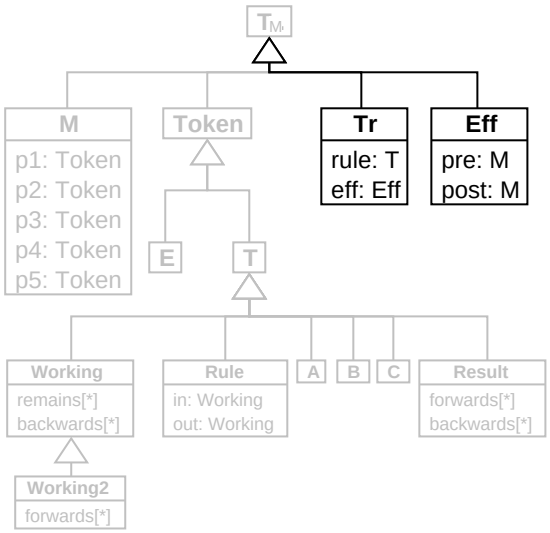

Abbildung 4.13: Das Transitionsregel-Typsystem zum Netz aus Abbildung 4.10.

Abbildung 4.13 zeigt das Typsystem $\mathrm{TS_r}$, das sich aus dem Markierungs-Typsystem $\mathrm{TS_m}$, dem Transitionsregeltyp Tr und dem Transitionswirkungstyp Eff ergibt.

Die abgeleitete Transitionsregel und die daraus resultierende Wirkung von t1 ist in Abbildung 4.14 dargestellt. Bei der Herleitung der abgeleiteten Transitionsregel findet sich im linken Term unter dem Feature rule die Transitionsregel von t1 wieder, die anschließend durch Bildung des Pfadwerts unter eff auf die betroffenen Ein- bzw. Ausgangsmarken aufgeteilt wird.

Die Wirkungs-Feature-Structure zeigt unter den Features pre bzw. post die Markierungs- bzw. Folgemarkierungs-Feature-Structures, die vom Typ M sind und für jede veränderte Stelle ein Feature aufweisen. Für die Werte dieser Features gibt es vier Möglichkeiten, die sich zum Teil nicht gegenseitig ausschließen:

1. Die entsprechende Stelle s befindet sich im Vorbereich der Transition t, nicht aber im Nachbereich. Das Feature für eine solche Stelle wird in $\mathrm{r_a}(t)@\mathsf{post}\, s$ mit

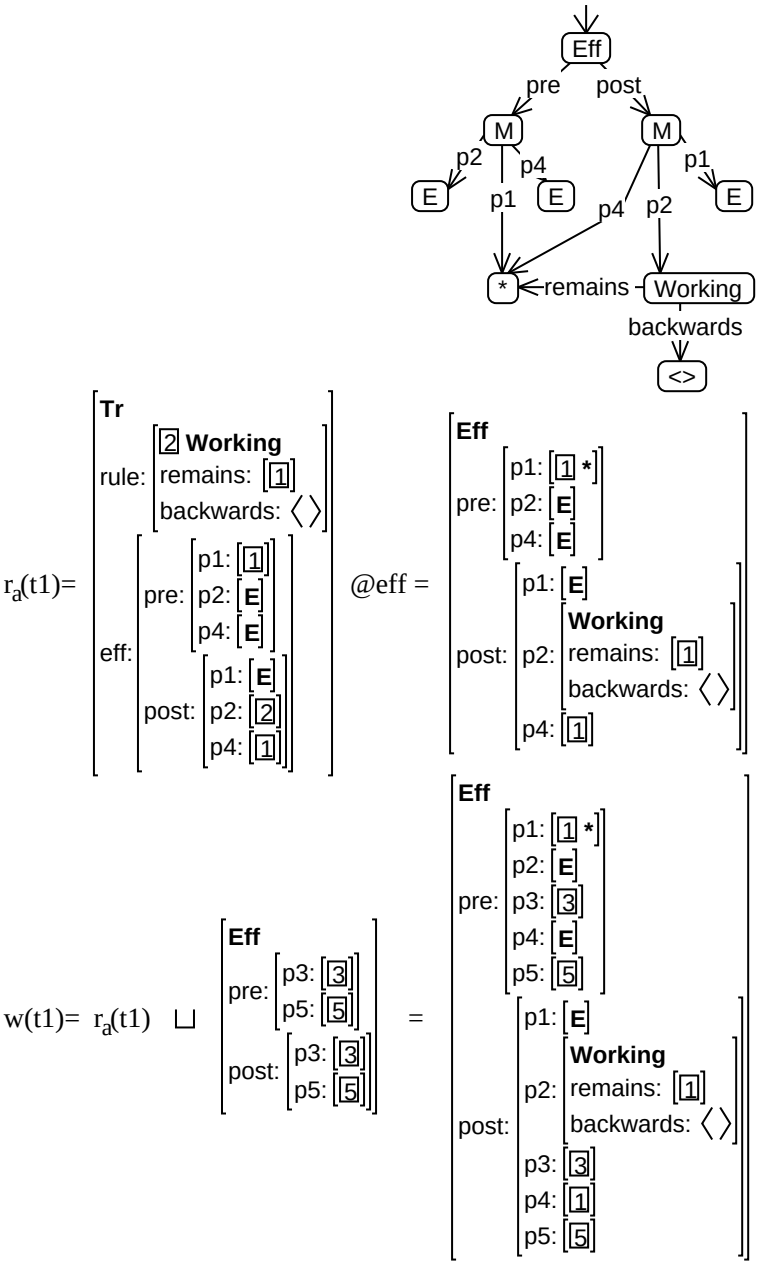

Abbildung 4.14: Abgeleitete Transitionsregel und Wirkung von `t1` aus dem Netz aus Abbildung 4.10. Die abgeleitete Transitionsregel ist zusätzlich in Graphnotation dargestellt.

dem speziellen Pfadwert [E] verbunden, um dafür zu sorgen, daß der Vorbereich nach dem Schalten leer ist. Für t1 ist p1 ein Beispiel für eine solche Stelle ($\mathsf{p1} \in \bullet\mathsf{t1} - \mathsf{t1}\bullet$), was nach Konstruktion zu $\mathrm{r_a}(\mathsf{t1})@\mathsf{post\,p1} \sim [\mathsf{E}]$ führt.

2. Die Stelle s befindet sich im Nachbereich der Transition t, nicht aber im Vorbereich. Das Feature für eine solche Stelle wird in $\mathrm{r_a}(t)@\mathsf{pre}\,s$ mit dem speziellen Pfadwert [E] verbunden, um dafür zu sorgen, daß der Nachbereich vor dem Schalten leer ist. p2 liegt nur im Nachbereich von t1 ($\mathsf{p2} \in \mathsf{t1}\bullet - \bullet\mathsf{t1}$), wodurch sich $\mathrm{r_a}(\mathsf{t1})@\mathsf{pre\,p2} \sim [\mathsf{E}]$ ergibt.

3. Die Stelle s liegt im Vorbereich oder im Nachbereich der Transition. Dann referenziert das entsprechende Feature den durch die Kantenanschrift gegebenen Knoten in der Transitionsregel. Ein Beispiel ist die Stelle p2, die im Nachbereich von t1 liegt ($\mathsf{p1} \in \bullet\mathsf{t1} \cup \mathsf{t1}\bullet$). Daraus entsteht nach Konstruktion die Strukturgleichheit $\mathrm{r_a}(\mathsf{t1})@\mathsf{rule\,remains} = \mathrm{r_a}(\mathsf{t1})@\mathsf{post\,p2}$.

4. Die Stelle s liegt weder im Vorbereich noch im Nachbereich der Transition t. Dann wird die Marke vom Vor- in den Nachbereich kopiert, da copymark dafür sorgt, daß $\mathrm{r_a}(t)@\mathsf{pre}\,s = \mathrm{r_a}(t)@\mathsf{post}\,s$. Ein Beispiel ist die Stelle p3, die weder im Vor- noch im Nachbereich von t1 liegt ($\mathsf{p3} \in \mathrm{S} - \bullet\mathsf{t1} - \mathsf{t1}\bullet$). Durch die Konstruktion wird eine Pfadgleichung etabliert, so daß $\mathrm{r_a}(\mathsf{t1})@\mathsf{pre\,p3} = \mathrm{r_a}(\mathsf{t1})@\mathsf{post\,p3}$.

Während Querverweise aus der Markierung in die Transitionsregel in der AVM-Notation zu zusätzlichen Knotenreferenzen führen, sind sie in der Graphnotation besonders deutlich als Koreferenzen zu erkennen. Die Überprüfung, ob eine Ausgangsstelle vor dem Schalten leer ist, sorgt dafür, daß nicht mehrere Marken auf einer Stelle liegen können, was für ein elementares FSNet nicht erlaubt ist.

Abbildung 4.15 zeigt nur die abgeleitete Transitionsregel von t2, da das Kopieren der übrigen Marken am vorherigen Beispiel deutlich geworden sein sollte und diese komplexere Regel zu unübersichtlich machen würde. Hier tritt mit p2 der interessante Fall einer Nebenbedingung auf. Obwohl dieser Fall in der Aufzählung oben nicht gesondert erwähnt wird, ergibt sich eine besondere Behandlung, da weder Fall 1 noch Fall 2 auf p2 und t2 zutreffen. Der Pfadwert [E] kommt hier nicht vor, da t2 weder reine Vorbedingungen noch reine Nachbedingungen aufweist.

Die abgeleitete Transitionsregel der weiteren Transitionen sind den bisher gezeigten sehr ähnlich und ergeben keine grundsätzlich neuen Fälle, weswegen diese hier nicht gezeigt werden.

Aus ihrer Wirkungen ergibt sich Aktivierung und Schalten von Transitionen in elementaren FSNets sehr einfach, wobei wir zunächst eine Referenzsemantik erhalten.

Definition 4.16 [Aktivierung, Schalten in elementaren FSNets nach

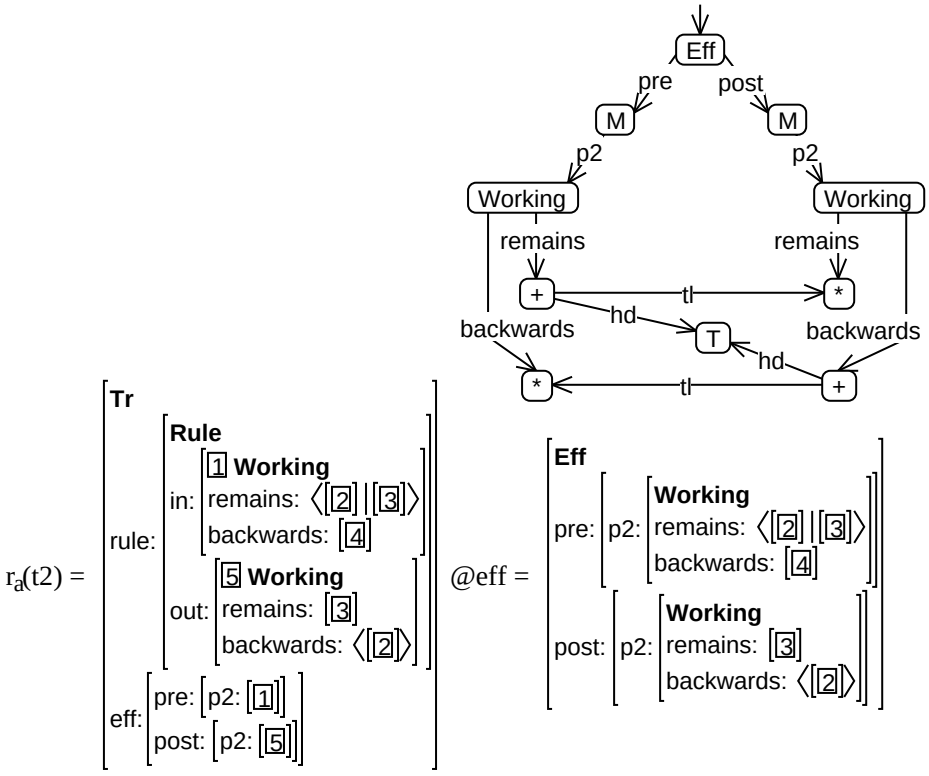

Abbildung 4.15: Die abgeleitete Transitionsregel von t2 aus dem Netz aus Abbildung 4.10 in AVM- und Graphnotation.

Referenzsemantik]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet, m_1, m_2 Markierungen von $EFSN$, $t \in \mathrm{T}$ eine Transition.

(a) t heißt *nach Referenzsemantik aktiviert*, falls:

$$\mathrm{w}(t) \sqcup \mathsf{pre}::\mathrm{m}_1 \text{ ist definiert}$$

(b) t *schaltet nach Referenzsemantik* m_1 zu m_2, symbolisch $\mathrm{m}_1 \xrightarrow[ref]{t} \mathrm{m}_2$, falls t in m_1 nach Referenzsemantik aktiviert ist und:

$$\mathrm{m}_2 = (\mathrm{w}(t) \sqcup \mathsf{pre}::\mathrm{m}_1)@\mathsf{post} \qquad \Diamond$$

Es handelt sich um eine Referenzsemantik, da die Markierung des gesamten Netzes als eine Feature Structure dargestellt wird und keine weiteren Einschränkungen

über diese Feature Structure angenommen wurden. Durch entsprechende Knotenreferenzen kann es somit vorkommen, daß das Schalten einer Transition t den Wert von Marken verändert, die sich weder im Vor- noch im Nachbereich von t befinden, sondern die lediglich eine Koreferenz zu einer von t verarbeiteten Marke oder deren Substrukturen aufweisen. Dies ist nur in einer Referenzsemantik möglich.

Um die Wertsemantik beim Schalten in Basis-FSNets zu rekonstruieren, muß die Markierung des elementaren FSNet sowohl zum Prüfen der Aktivierung als auch zum Schalten zuvor in eine Wertmarkierung umgewandelt werden. Dafür wird die oben definierte Funktion valmark verwendet, die aus einer (Referenz-)Markierung eine Wertmarkierung erzeugt.

Definition 4.17 [Aktivierung, Schalten in elementaren FSNets nach Wertsemantik]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet, m_1, m_2 Markierungen von $EFSN$, $t \in \mathrm{T}$ eine Transition.

(a) t heißt *nach Wertsemantik aktiviert*, falls:

$$\mathrm{w}(t) \sqcup \mathsf{pre}{::}\mathrm{valmark}(\mathrm{m}_1) \text{ ist definiert}$$

(b) t *schaltet nach Wertsemantik* m_1 zu m_2, symbolisch $\mathrm{m}_1 \xrightarrow[val]{t} \mathrm{m}_2$, falls t in m_1 nach Wertsemantik aktiviert ist und:

$$\mathrm{m}_2 = \mathrm{valmark}((\mathrm{w}(t) \sqcup \mathsf{pre}{::}\mathrm{valmark}(\mathrm{m}_1))@\mathsf{post}) \qquad \Diamond$$

Eine andere Variante ist, die Anfangsmarkierung m_0 in eine Wertmarkierung umzuwandeln. Wenn dann ausschließlich die Wertsemantik-Schaltregel benutzt wird, muß die valmark-Funktion nur beim Schalten (nicht bei der Überprüfung auf Aktivierung) und dort auch nur auf das Ergebnis der Folgemarkierung angewandt werden. Da somit sichergestellt ist, daß sich das Netz nur in Wertmarkierungen befinden kann, kann die Umwandlung der aktuellen Markierung entfallen (sie schadet aber auch nicht, da offensichtlich für jede Wertmarkierung m gilt: $\mathrm{valmark}(m) = m$).

In elementaren FSNets können also beide Arten der Schaltsemantik sehr ähnlich beschrieben werden. Dies legt den Ansatz nahe, innerhalb eines Netzes beide Semantiken zur Verfügung zu stellen, worauf wir in Abschnitt 4.2.5 zurückkommen.

Im folgenden Abschnitt wird, analog zur universellen Turingmaschine, ein universelles EFSNet vorgestellt, das beliebige andere EFSNets ausführen kann.

4.2.4 Universelle elementare FSNets

Im vorherigen Abschnitt wurde eine Schaltfolgensemantik für elementare FSNets vorgestellt, welche die Aktivierung und das Schalten von Transitionen durch Anwendungen von abgeleiteten Transitionsregeln definiert hat. Es soll nun gezeigt werden,

daß elementare FSNets genügend ausdrucksstark sind, um ihre eigene Schaltsemantik zu simulieren. Dazu konstruieren wir universelle EFSNet, die andere EFSNets simulieren können.

Die Idee einer universellen Maschine ist für Turingmaschinen bekannt ([Herken 1988]). Eine universelle Turingmaschine erhält als Eingabe die Kodierung einer beliebigen Turingmaschine sowie deren Eingabe. Das Verhalten der codierten Turingmaschine wird von der universellen Turingmaschine simuliert, so daß genau die gleiche Ausgabe entsteht wie bei einer direkten Ausführung der simulierten Turingmaschine.

Um EFSNets von einem EFSNet simulieren zu lassen, müssen wir also zunächst eine vollständige Kodierung eines EFSNets durch Feature Structures finden. Dies fällt nicht schwer, da in EFSNets sowohl Markierungen (Definition 4.7) als auch die Wirkung von Transitionen (Definition 4.15) durch Feature Structures dargestellt werden. Zusätzlich wird ein Objekt benötigt, das diese Information in einer Feature Structure zusammenfaßt.

Wir benötigen dafür ein Typsystem, das dem Transitionsregel-Typsystem aus Definition 4.12 ähnelt. Damit die Kodierung besser von einem universellen EFSNet verarbeitet werden kann, wird die Repräsentation von Markierungen und damit die Konstruktion der abgeleiteten Transitionsregeln und Wirkungen leicht modifiziert.

Einzelne Marken werden wie zuvor durch den Typ Token mit den Subtypen E für die leere Markierung und dem allgemeinsten Typ des Typsystems des zu simulierenden Netzes dargestellt. Statt eines Typs M für eine Markierung wird eine totale Ordnung über der Stellenmenge S definiert und eine Listendarstellung gewählt, so daß Markierungen als Listen des Typs Token$*$ der Länge |S| repräsentiert werden.

Der Typ Eff zur Darstellung der Transitionswirkungen wird entsprechend modifiziert, so daß seine Features pre und post nicht von Typ M, sondern vom Typ Token$*$ sind. Die abgeleiteten Transitionsregeln werden analog zu Definition 4.13 konstruiert, nur daß für eine Stelle $s_i \in$ S, die das i-te Element in der Ordnung über S ist, statt des Features s_i der Pfad $\mathsf{tl}^{i-1}\,\mathsf{hd}$ benutzt wird. Die Wirkungen ergeben sich durch Anwendung einer analog konstruierten copymark-Funktion (siehe Definition 4.14).

Zur Repräsentation des gesamten Netzes als eine Feature Structure dient schließlich ein Typ EFSNet mit den Features m0 vom Typ Token$*$ und rules vom Typ Eff$*$. m0 repräsentiert die Anfangsmarkierung, während rules eine Liste mit den Wirkungen aller Transitionen des Netzes enthält.

Das so erhaltene Typsystem wird *universelles EFSNet-Typsystem* eines gegebenen EFSNets genannt. Die Verwendung von Listen als Mengen hat den (hier unwesentlichen) Nachteil, daß eine Reihenfolge der Elemente impliziert wird. Dadurch ist die hier gewählte Kodierung nicht eindeutig, sondern hängt von den für S und T gewählten Ordnungen ab. Die Eindeutigkeit von Kodierungen ist aber nur dann von Interesse, wenn diese verglichen (oder gar unifiziert) werden sollen, was im folgenden nicht betrachtet wird.

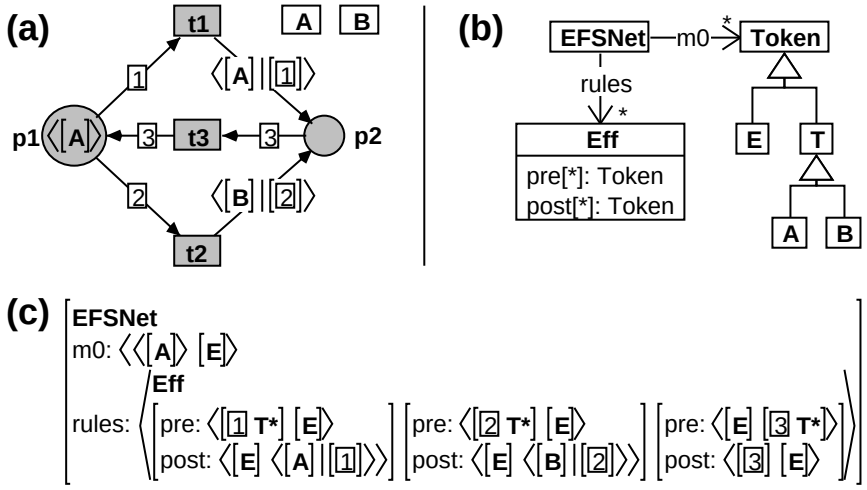

Abbildung 4.16: Ein einfaches EFSNet mit seinem Typsystem in (a), dessen universelles EFSNet-Typsystem in (b) und die Kodierung des Netzes als Feature Structure über diesem Typsystem in (c).

Abbildung 4.16 zeigt ein einfaches EFSNet mit seinem Typsystem in (a), dessen universelles EFSNet-Typsystem in (b) und die Kodierung des Netzes als Feature Structure über diesem Typsystem in (c). Als Reihenfolge für die Kodierung in (c) wurde für die Stellen $\langle \mathsf{p1}, \mathsf{p2} \rangle$ und für die Transitionswirkungen $\langle \mathsf{t1}, \mathsf{t2}, \mathsf{t3} \rangle$ gewählt.

Das Beispielnetz in (a) ergänzt die als Anfangsmarkierung in p1 vorhandene Liste mit dem Element [A] vorne durch ein weiteres [A], falls t1 schaltet, oder durch ein [B], falls t2 schaltet und legt die ergänzte Liste auf p2 ab. Sodann kann t3 die Liste von p2 nach p1 zurücktransportieren. Es werden also im Verlauf des Schaltens Listen von [A]s und [B]s erzeugt, die (aufgrund der Anfangsmarkierung) immer mit einem [A] enden, ansonsten aber beliebige Elemente enthalten.

Ein EFSNet, das die Kodierung eines EFSNets simulieren soll, muß als zentrale Funktion eine der Transitionswirkungen auf die aktuelle Markierung anwenden. Nach Wertsemantik muß danach die Funktion valmark angewandt werden, was sich in einem universellen EFSNet simulieren läßt, indem die Markierungs-Feature-Structure in eine Feature Structure für jede Marke zerlegt und anschließend wieder zusammengefügt wird[4]. Referenzsemantik ist demnach durch ein EFSNets einfacher zu simulieren, weshalb wir in den folgenden Ausführungen mit dieser Semantik beginnen. Die weiter unten gezeigten universellen EFSNets selbst müssen hingegen nach Wertsemantik ausgeführt werden, da Transitionsregeln mehrfach angewandt werden müssen, ohne dabei verändert zu werden. Dies ist nur durch Kopieren möglich, was wiederum nur in der Wertsemantik möglich ist. Ein universelles

[4]Für die Idee für diese Konstruktion möchte ich mich an dieser Stelle bei Olaf Kummer bedanken.

EFSNet kann sich also insbesondere selbst simulieren (wie auch die universelle Turingmaschine).

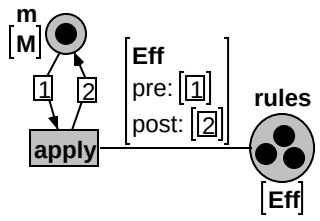

Abbildung 4.17: Anwendung einer Transitionswirkung auf eine Markierung als FSNet modelliert.

Abbildung 4.17 zeigt ein FSNet, in dem angenommen wird, daß die aktuelle Markierung des zu simulierenden EFSNets in der Stelle m und die aus dessen Transitionsregeln abgeleiteten Transitionswirkungen als Marken in der Stelle rules liegen. Es wird eine im Basis-FSNet nicht formal eingeführte Testkante zur Stelle rules benutzt, damit die dort liegenden Wirkungen beim Schalten nicht verändert werden. Die Semantik der Testkante sollte sich in diesem Fall erschließen; das Beispiel dient ohnehin nur zur Herleitung der unten gezeigten universellen EFSNets.

Die Transition apply in Abbildung 4.17 ist nach der Schaltregel für FSNets in genau sovielen Schaltmodi aktiviert, wie Transitionswirkungen auf die aktuelle Markierung anwendbar sind. Für jede anwendbare Transitionswirkung ist nach Definition 4.16 (a) die entsprechende Transition im zu simulierenden EFSNet aktiviert. Beim Schalten wird einer der Schaltmodi nichtdeterministisch ausgewählt und die dadurch selektierte Wirkung durch Unifikation mit den Kantenausdrücken und den gebundenen Marken angewandt. Dies entspricht genau der in Definition 4.16 (b) gegebenen Schaltregel.

Soll die Anwendung einer Wirkung als *elementares* FSNet dargestellt werden, muß ein Modell gefunden werden, das mit höchstens einer Marke pro Stelle auskommt und keine Testkanten verwendet. Wir lösen das erste Problem, indem wir wie in der oben angegebenen Kodierung alle Transitionswirkungen zu einer Liste in einer Marke zusammenfassen. Das Netz muß nun diese Liste durchsuchen, um eine aktivierte Transition zu finden und zu schalten. Das zweite Problem wird beseitigt, indem von der Marke mit der Liste der Wirkungen jeweils eine Kopie erzeugt und bei der Anwendung verbraucht wird. Außerdem wird eine Transition ergänzt, welche die gegebene Kodierung des zu simulierenden Netzes in Markierung und Liste der Wirkungen zerlegt.

Abbildung 4.18 zeigt das resultierende universelle EFSNet, das je nach Anfangsmarkierung in der Stelle efsn Schaltfolgen jedes beliebigen EFSNets nach Referenzsemantik simulieren kann. Wie oben bereits gesagt, ist das universelle EFSNet selbst

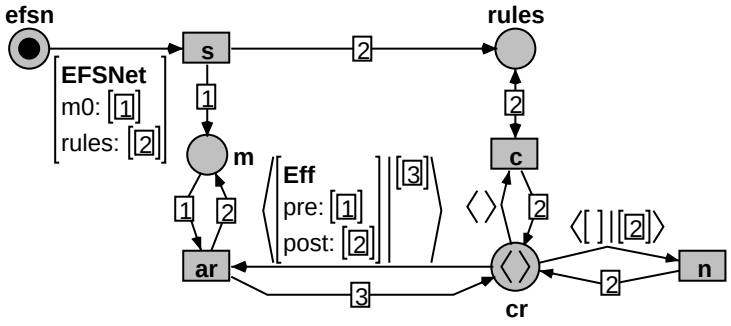

Abbildung 4.18: Ein universelles EFSNet, das beliebige EFSNets nach Referenzsemantik ausführen kann.

nach Wertsemantik auszuführen.

Transition s (*split*) zerlegt die EFSNet-Kodierung in Anfangsmarkierung und Liste der Transitionswirkungen. Die Anfangsmarkierung wird in die Stelle m gelegt, welche die jeweils aktuelle Markierung enthält. Die Liste der Transitionswirkungen wird in der Stelle rules gespeichert.

Transition c (*copy*) ersetzt eine leere aktuelle Liste von Transitionswirkungen in Stelle cr (*current rules*) durch eine neue Kopie der vollständigen Liste aus Stelle rules. c kann aufgrund der leeren Liste als Anfangsmarkierung in cr gleich nach s einmal schalten.

Transition ar (*apply reference semantics rule*) übernimmt die Rolle der apply-Transition aus Abbildung 4.17 und verändert somit die aktuelle Markierung in m. Im Unterschied zu apply wird die Transitionswirkung aus dem Kopf der Liste der aktuellen Transitionswirkungen in cr gelesen (man beachte die leicht veränderte Kantenanschrift) und statt zu testen (siehe oben) die gesamte Liste konsumiert und durch ihre Restliste (*tail*) ersetzt. Es besteht keine Gefahr, daß die Restliste bei der Unifikation verändert wurde, da zwischen Transitionswirkungen nach Konstruktion keine Koreferenzen bestehen können.

Die Transition n (*next*) dient schließlich der Abarbeitung der Liste der Wirkungen. Falls der Kopf der aktuellen Restliste die Wirkung einer Transition enthält, die nicht aktiviert ist, kann nur n schalten. Fall die Transition aktiviert ist, können sowohl ar als auch n schalten. Es läßt sich mit einem EFSNet nicht modellieren, daß nur dann zur nächsten Transitionswirkung übergegangen werden soll, wenn im aktuellen Zustand ar *nicht* aktiviert ist. Dafür müßte man ausdrücken können, daß eine Transition aktiviert ist, wenn bestimmte Feature Structures *nicht* unifizierbar sind. Diese Modellierung ist aber auch gar nicht gewünscht, wie im folgenden begründet wird.

Das universelle EFSNet soll alle Schaltfolgen simulieren können, die das simu-

lierte Netz nach Schaltfolgensemantik durchlaufen könnte. Wenn also immer die nächste aktivierte Transition auch sofort geschaltet werden würde, kämen einige Schaltfolgen in der Simulation nie zustande. Auch mit der vorliegenden Modellierung können unerwünschte Schaltfolgen des universellen EFSNets ablaufen, die aber durch zusätzliche Bedingungen ausgeschlossen werden können. Da der Konflikt zwischen ar und n nichtdeterministisch gelöst wird, ist es beispielsweise möglich, daß immer nur n schaltet und damit das simulierte Netz stehenbleibt, obwohl dort noch Transitionen aktiviert sind. Um eine solche „unfaire“ Lösung von Konflikten zu verhindern, gibt es für Petrinetze die *verschleppungsfreie* und die *faire* Schaltregel ([Jessen und Valk 1987]). Ein Netz schaltet demnach *nicht* verschleppungsfrei (bzw. fair), wenn von einer erreichten Markierung ab eine unendliche Schaltfolge schaltet, bei der eine Transition zwar permanent, d.h. in allen durchlaufenen Markierungen (bzw. unendlich oft) aktiviert ist, aber nie schaltet ([Jessen und Valk 1987], S. 232).

Damit das simulierte Netz verschleppungsfrei geschaltet wird, muß das universelle EFSNet fair geschaltet werden. Verschleppungsfreies Schalten des universellen EFSNets reicht dafür nicht aus, da ar beim zyklischen Durchlaufen der Transitionswirkungen nicht permanent aktiviert ist (selbst, wenn *alle* Transitionen des simulierten Netzes aktiviert sind, ist ar für die leere Liste nicht aktiviert). Die unendliche Schaltfolge, in der ar unendlich oft aktiviert ist, aber nie schaltet, wird erst durch die faire Schaltregel ausgeschlossen. Ein faires Schalten des simulierten Netzes kann für den allgemeinen Fall durch ein universelles EFSNet höchst wahrscheinlich nicht erreicht werden, dies müßte aber noch bewiesen werden.

Transition ar könnte statt der Restliste (Knotenreferenz $\boxed{3}$) auch die leere Liste zurücklegen. Bei entsprechender Lösung des Konflikts zwischen ar und n könnten immer noch alle Schaltfolgen des simulierten Netzes erzeugt werden. Selbst wenn das universelle EFSNet fair schaltet, wäre mit dieser kleinen Änderung des Modells aber die Verschleppungsfreiheit des simulierten Netzes nicht mehr garantiert. Man stelle sich als Gegenbeispiel ein Netz mit dem Transitionsvektor $\langle \mathsf{e1}, \mathsf{e2}, \mathsf{e3} \rangle$ vor, in dem e1 nie aktiviert ist, e2 und e3 dagegen permanent. Ein solches Netz könnte vom modifizierten universellen EFSNet durch die faire Schaltfolge $s\,(\mathsf{c}\ \mathsf{n}\ \mathsf{ar})^\omega$ simuliert werden, was für das simulierte Netz die Schaltfolge $\mathsf{e2}^\omega$ ergibt, die nicht verschleppungsfrei ist, da e3 permanent aktiviert ist, aber nie schaltet.

Betrachten wir schließlich eine weitere Eigenschaft der Netze: Verklemmungen (*deadlocks*). Ein Netz heißt verklemmt, wenn in der aktuellen Markierung keine Transition aktiviert ist. Falls das simulierte Netz verklemmt ist, kann das universelle EFSNet weiterhin die Schaltfolge $\mathsf{c}\ \mathsf{n}^{|S|}$ beliebig oft ausführen. Verklemmung stimmt also bei simuliertem und universellem EFSNet nicht überein. Wenn das universelle EFSNet fair schaltet, kann eine Verklemmung des simulierten Netzes aber daran erkannt werden, daß die unendliche Schaltfolge $(\mathsf{c}\ \mathsf{n}^{|S|})^\omega$ als Postfix auftritt, was durch die Fairneß für nicht verklemmende simulierte Netze ausgeschlossen ist. Eine Modellierung, in der Verklemmung bei simuliertem und simulierendem Netz übereinstimmt, scheint mit einem EFSNet nicht möglich. Mit dem in Abbildung 4.17

skizzierten FSNet sollte eine solche Eigenschaft erreicht werden können, dies wurde hier aber nicht untersucht.

Das bisher vorgestellte „universelle“ EFSNet kann nur EFSNets nach Referenzsemantik simulieren und verdient damit diesen Namen noch nicht wirklich. Nachdem am Beispiel der Referenzsemantik die Simulation von EFSNets ausführlich diskutiert wurden, betrachten wir deshalb nun die Simulation der Wertsemantik.

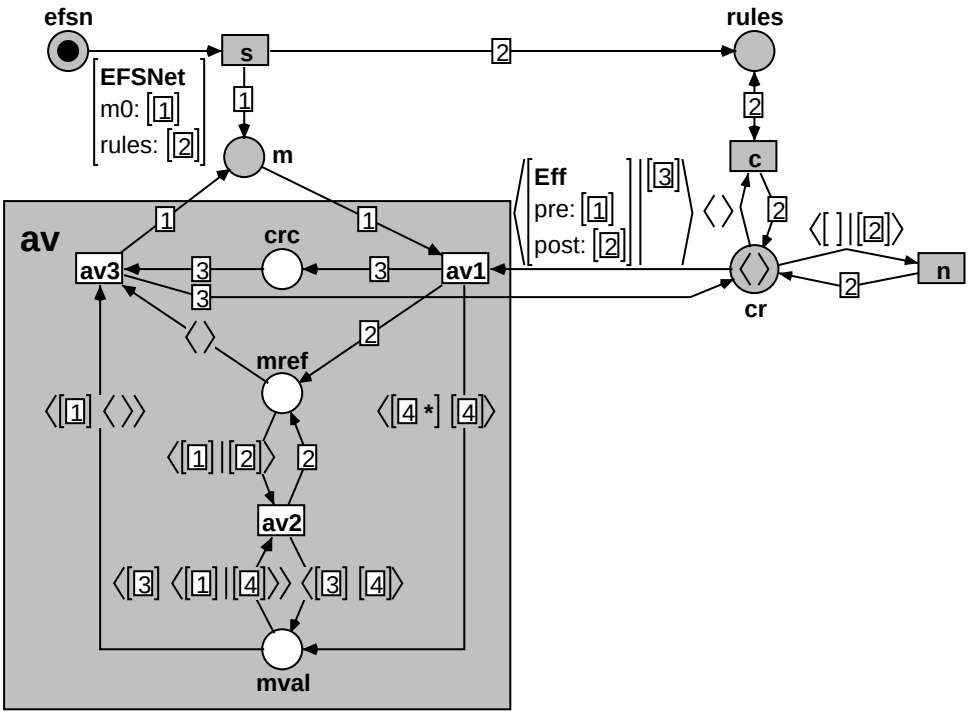

Abbildung 4.19: Ein universelles EFSNet, das beliebige EFSNets nach Wertsemantik ausführen kann.

Die Problematik der Wertsemantik wurde oben bereits angesprochen: Nach Anwendung der Transitionswirkung müssen Koreferenzen zwischen Marken aufgetrennt werden. Im EFSNet-Formalismus werden Koreferenzen genau dann aufgetrennt, wenn die Substrukturen nach Wertsemantik in verschiedenen Marken abgelegt werden. Das universelle EFSNet schaltet selbst nach Wertsemantik. Wir müssen das Modell demnach so ändern, daß die Liste aller Marken in einzelne Marken zerlegt und zu einer neuen Liste zusammengesetzt wird. Dies kann durch eine *Transitionsverfeinerung* ([Jessen und Valk 1987]) von ar geschehen, die in Abbildung 4.19 als grauer Kasten mit der Anschrift av (*apply value semantics rule*) zu sehen ist.

Das Kopieren der Marken-Liste wird in drei Schritten vorgenommen: Initialisierung (Transition av1), Schleife (av2) und Schleifenende (av3). Dabei sind einige

Fallen zu beachten, die leicht zu unerwünschtem Verhalten führen könnten.

Transition av1 initialisiert die Stellen mref und mval und speichert die Restliste der Transitionswirkungen auf der Stelle crc (*current rules copy*). Würde die Restliste sofort auf die Stelle cr zurückgelegt, gilt das in der Diskussion der oben erwähnten Modifikation des universellen EFSNets gesagte: Es würden zwar noch alle Schaltfolgen simuliert werden können, faires Verhalten des universellen Netzes würde aber nicht mehr zu verschleppungsfreiem Verhalten des simulierten Netzes führen. Als Gegenbeispiel kann dasselbe Netz wie oben herangezogen werden. Wenn im universellen Netz Transition n zweimal schaltet, *während* die Markierung kopiert wird, entsteht der oben beschriebene Fall. Um dies zu verhindern, wird die Restliste erst nach dem Kopieren von Transition av3 zurückgelegt.

Die Stelle mref enthält die Restliste der Markierung nach Referenzsemantik und wird entsprechend von Transition av1 mit der direkt aus der Anwendung der Transitionswirkung hervorgegangenen Folgemarkierung (Knotenreferenz [2]) initialisiert. In der Stelle mval soll die Wertmarkierung konstruiert werden. Hier würde in einem weniger optimierten Modell mit einer leeren Liste begonnen und die Elemente der Liste in mval sukzessive in die Liste auf mref übernommen. Da bei einfach verketteten Listen auf einfache Weise nur vor der Liste Elemente eingefügt werden können, würde dabei aber gleichzeitig eine Umkehrung der Liste entstehen, die hier nicht erwünscht ist und durch weitere Transitionen und Stellen wieder rückgängig gemacht werden müßte. Deshalb wurde statt der skizzierten Modellierung die folgende gewählt: In mval wird eine *offene* Liste (siehe Abschnitt 2.3.5) mit einer Referenz auf ihr offenes Ende abgelegt. Offene Liste und Listenende-Referenz werden als Liste der Länge zwei[5] zu einer Feature Structure zusammengefaßt. Als Initialwert zeigen beide auf dieselbe leere offene Liste.

Das Kopieren einer einzelnen Marke wird durch Transition av2 vorgenommen. Die Marke wird aus dem Kopf der Liste von mref (Knotenreferenz [1]) gelesen und am Ende der Liste auf mval angehängt, indem sie über die Listenende-Referenz in die Liste (Knotenreferenz [3]) hinein unifiziert wird. In mval wird ein „Paar“ aus derselben Liste und einer Listenende-Referenz auf die Restliste hinter dem neu eingefügten Element (Knotenreferenz [4]) abgelegt. In mref wird die Restliste der Referenzsemantik-Markierung (Knotenreferenz [2]) abgelegt.

Der Kopiervorgang wird durch Transition av3 beendet, wenn in mref die leere Liste vorgefunden wird. In mval befindet sich dann die entsprechende Wertmarkierung als offene Liste. Durch Unifikation mit dem Ausdruck an der Kante von mval nach av3 wird die offene Liste über die Listenende-Referenz geschlossen und über Knotenreferenz [1] in die Stelle m zurückgelegt. Gleichzeitig wird wie oben besprochen die Restliste der Transitionswirkungen (Knotenreferenz [3]) nach cr verschoben.

Wir haben nun die Simulation von EFSNets mit zwei recht ähnlichen „univer-

[5] Hier hätten auch die in Abschnitt 2.3.5 eingeführten Tupel verwendet werden können. Da diese im Rest der Arbeit aber nicht zum Einsatz kommen und der Leser daher mit der Listenschreibweise vertrauter ist, wird eine Liste konstanter Länge benutzt.

sellen“ EFSNets nach Referenz- bzw. Wertsemantik modelliert. Als Abschluß dieses Abschnitts soll gezeigt werden, daß es tatsächlich *ein* universelles EFSNet gibt, das beide Semantiken zur Simulation anbietet.

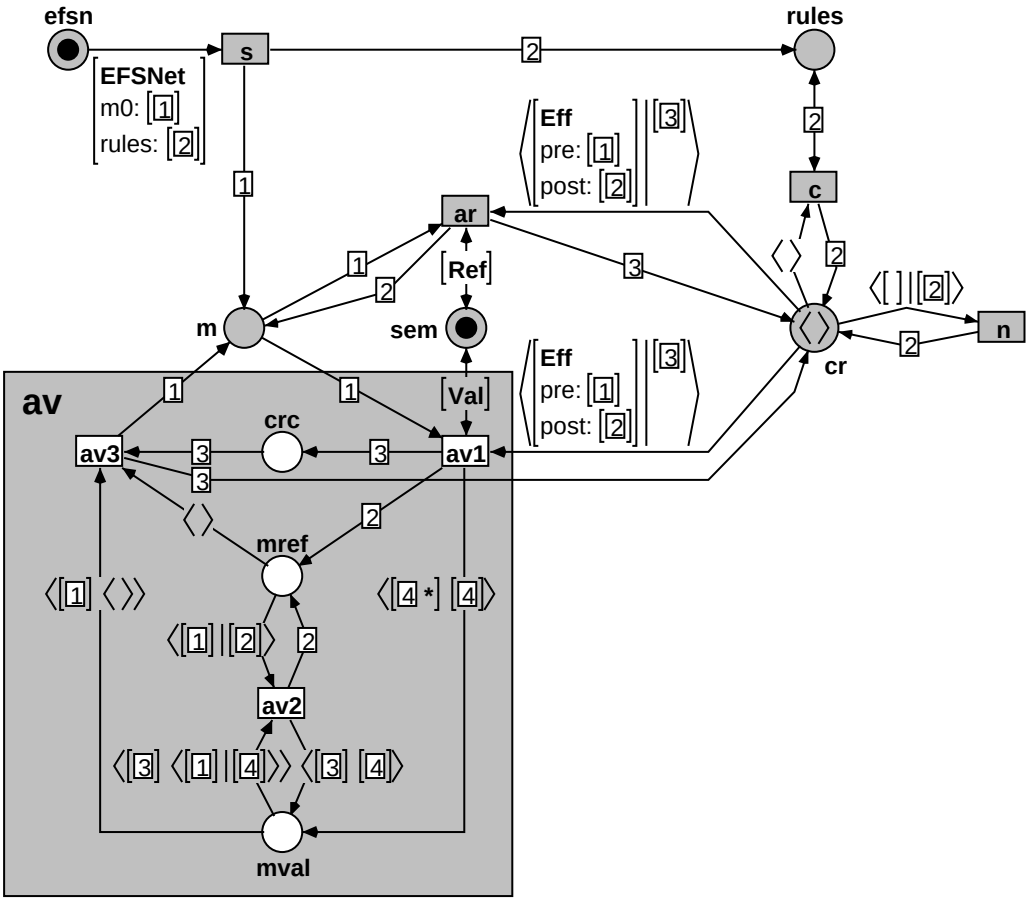

Abbildung 4.20: Ein universelles EFSNet, das beliebige EFSNets nach Wert- oder Referenzsemantik ausführen kann.

Abbildung 4.20 zeigt ein Modell eines solchen wirklich universellen EFSNets, das beliebige EFSNets nach Wert- oder Referenzsemantik simulieren kann. Dieses Netz enthält beide Versionen der apply-Transition und aktiviert durch eine Nebenbedingung die mit der gewünschten Semantik. Im gezeigten Modell wird davon ausgegangen, daß es im oben eingeführten universellen EFSNet-Typsystem einen Typ Sem mit den disjunkten Subtypen Val und Ref gibt. Durch die Anfangsmarkierung der Stelle sem entscheidet man, ob das in efsn gegebene Netz nach Wert- ([Val]) oder Referenzsemantik ([Ref]) simuliert werden soll. Diese Information könnte alternativ als Teil der Netzspezifikation modelliert werden, indem der oben eingeführte Typ

EFSNet ein zusätzlichen Feature sem vom Typ Sem erlaubt. Die Anfangsmarkierung der Stelle sem würde entfallen und durch eine eingehende Kante von Transition s aus ersetzt, die den Wert des Features sem in der Netzspezifikation selektiert.

Zusammenfassend ist zu sagen, daß es gelungen ist, die Schaltsemantik bestimmter FSNets wiederum mit einem FSNet zu beschreiben, wenn auch nur für den eingeschränkten Formalismus der elementaren FSNets. Dies konnte durch die konsequente Verwendung eines Basisformalismus, in diesem Fall Feature Structures, erreicht werden. Gerade bei einem Ansatz, der auf mehreren verschiedenen Formalismen basiert (hier Petrinetze, Typsysteme und Logik), ist eine solche Vereinheitlichung erstrebenswert, um mit einer möglichst überschaubaren Menge von Konzepten auszukommen. Auch Farwer definiert Linearlogische Petrinetze (siehe Abschnitt 3.3.2) zunächst als Petrinetze mit linearlogischen Formeln als Marken, zeigt aber später, daß auch das Systemnetz selbst und damit das gesamte Objektsystem durch Lineare Logik beschrieben werden kann.

Für die Simulation von Basis-FSNets durch ein Basis-FSNet fehlt wie in Abschnitt 4.2 diskutiert eine direkte Darstellung von Mengen als Feature-Werte.

Nach diesem theoretischen Exkurs wenden wir uns einer Erweiterung der Schaltsemantik zu, die in verteilten Systemen praktische Relevanz hat: Der gleichzeitigen Verwendung von Wert- und Referenzsemantik in einem Netz.

4.2.5 Kombination von Wert- und Referenzsemantik

In [Valk 1999] wird in bezug auf Wert- und Referenzsemantik ein weiteres Phänomen betrachtet. Bisher wurde davon ausgegangen, daß für ein gesamtes Netz dieselbe Semantik gilt. Auch wenn man mit Petrinetzen physikalische Verteilung modellieren kann, so muß nicht unbedingt jede Stelle einen anderen Ort darstellen. Vielmehr scheint es sinnvoll, die Stellen eines Netzes in Mengen von Stellen (Partitionen) einzuteilen, die als zueinander lokal angesehen werden, wogegen Stellen aus unterschiedlichen Partitionen als entfernt betrachtet werden.

In einer solchen Modellierung bietet es sich an, innerhalb von Partitionen Referenzsemantik zu benutzten, zwischen Partitionen aber Wertsemantik zu fordern. Dies entspricht beispielsweise der Unterscheidung zwischen Mehrprozessorarchitekturen mit gemeinsamem Speicher und verteilten Systemen, die ausschließlich über Nachrichten kommunizieren können.

Abbildung 4.21 zeigt ein Petrinetz, bei dem die Einteilung der Stellen in verschiedene Partitionen durch grau hinterlegte Flächen angegeben ist. Für die Notation solcher Stellenparitionierungen könnte überlegt werden, auf UML-Verteilungsdiagramme (siehe Abschnitt 2.2.3 zurückzugreifen. Dieser Ansatz wird aber in dieser Arbeit nicht weiter verfolgt.

Die Partitionierung der Stellen hat Einfluß auf die Semantik der Transitionen: Während beispielsweise t1 zwei Referenzen auf denselben Wert als Marken in p2 und p3 ablegt, erzeugt t2 eine Kopie einer Marke von p2 in q1, da p2 und q1 in

unterschiedlichen Partitionen liegen. Als Konsequenz können von t3 vorgenommene Änderungen die Aktivierung von t2 beeinflussen. Auch bei einer Vereinigung (*join*) wie t6 verhalten sich Stellen, die in derselben Partition liegen, anders als solche in unterschiedlichen Partitionen. Wenn beispielsweise q3 und q4 dieselbe Marke referenzieren, kann die Unifikation trivial sein (die Marken sind bereits identisch). Bei Marken von Stellen aus verschiedenen Partitionen, wie p4 und q4, müssen dagegen deren Werte unifiziert werden.

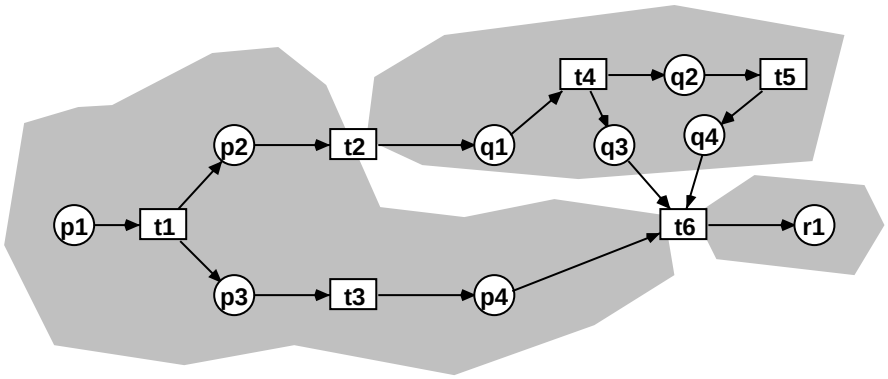

Abbildung 4.21: Ein Petrinetz, dessen Stellen in Partitionen unterteilt sind. Innerhalb der Partitionen wird Referenzsemantik angenommen, zwischen Partitionen Wertsemantik.

Mathematisch betrachtet ist eine Stellen-Partitionierung nichts anderes als eine reflexive, transitive und symmetrische Relation, also eine Äquivalenzrelation, die eine Menge von Stellen in Äquivalenzklassen einteilt.

Definition 4.18 [Stellen-Partitionierung]
Eine *Stellen-Partitionierung* local_S ist eine Äquivalenzrelation über einer Menge von Stellen S. ◊

Wie bei der Wertmarkierung definieren wir die Bedingung, unter der eine Elementarmarkierung nach einer solchen Semantik als wohlgeformt angesehen wird.

Definition 4.19 [Partitionsmarkierung]
Sei m_p eine Elementarmarkierung über einem Typsystem $\text{TS} = \langle \text{Type}, \sqsubseteq, \text{Feat}, \text{approp} \rangle$ und einer Stellenmenge S, $\text{MTS}_{\text{TS,S}}$ das Markierungs-Typsystem zu TS und S und local_S eine Stellen-Partitionierung über S. $\text{m}_\text{p} \in \text{MTS}_{\text{TS,S}}$ heißt eine *Partitionsmarkierung* genau dann wenn gilt:

$$\forall s, s' \in \text{S}, \pi, \pi' \in \text{Feat}^* : \text{m}_\text{p}@(s\,\pi) = \text{m}_\text{v}@s'\,\pi' \Rightarrow s \; \text{local}_\text{S} \; s'$$

(zeigen zwei Pfade auf dieselbe Unterstruktur, so müssen die Unterstrukturen in Marken derselben Partition liegen). ◊

Eine Partitions-Schaltsemantik ergibt sich analog zur Wertsemantik, indem man die valmark-Funktion durch eine Funktion ersetzt, die nur Unterstrukturen kopiert, die in unterschiedlichen Partitionen liegen. Eine technisch günstigere Definition einer Partitionsmarkierung erhält man, wenn man die Definition von Elementarmarkierungen leicht modifiziert: Eine Markierungs-Feature-Structure erhält zunächst für jede Partition ein Feature und erst die Partitionsknoten zeigen mit Features für ihre Stellen auf die Markenwerte. Auch das in Abschnitt 4.2.4 vorgestellte universelle EFSNet kann mit dieser Darstellung von Partitionen leicht an die Partitionssemantik angepaßt werden. Aufgrund der starken Ähnlichkeit werden die Definitionen für diese modifizierte Markierung und die Partitions-Schaltsemantik sowie das modifizierte universelle EFSNet hier nicht wiedergegeben.

Die in den vorhergehenden Abschnitten beschriebenen Semantiken ergeben sich als Sonderfälle der Partitionssemantik: Wird für jede Stelle des Netzes eine einzelne Partition gewählt, so erhält man Wertsemantik, werden dagegen alle Stellen einer einzigen Partition zugeordnet, so erhält man Referenzsemantik.

Es gibt noch weitere Möglichkeiten, Wert- und Referenzsemantik in einem Netz gemeinsam zu benutzen, die Gegenstand weiterer Forschung sind. Zum einen kann man für die Partitionsrelation beliebige symmetrische Relationen an Stelle von Äquivalenzrelationen betrachten. Dies führt statt einer Einteilung der Stellenmenge in Gruppen zu einer fallweisen Entscheidung, ob zwei Stellen „lokal“ zueinander sind. Zum anderen kann man die Unterscheidung zwischen Wert- und Referenzsemantik statt den Stellen den Transitionen zuordnen. Man unterscheidet dann zwischen Transitionen, die existierende Marken modifizieren (Referenzsemantik), und solchen, die modifizierte Kopien von Marken produzieren (Wertsemantik). Ein Beispiel für eine solche gleichzeitige Verwendung von Wert- und Referenzsemantik wird in Abschnitt 4.3.2 bei der Konstruktion von Prozessen von Netzen in Netzen diskutiert.

4.2.6 Eine Schrittsemantik für elementare FSNets

Mit Hilfe der in Abschnitt 4.2.3 definierten Schaltfolgensemantik kann man analog zu anderen Petrinetz-Formalismen die Menge der erreichbaren Markierungen für ein elementares FSNet bestimmen (siehe nächster Abschnitt). Wir wollen im folgenden aber auch weitere Semantiken vorstellen: Die Schrittsemantik erlaubt das gleichzeitige Schalten mehrerer Transitionen, während die Prozeßsemantik die Ausführung eines Netzes echt nebenläufig beschreibt.

Eine Schrittsemantik wurde für einfache Petrinetze bereits in Kapitel 3.2.2 erwähnt. Analog soll es in elementaren FSNets möglich sein, eine Menge von kontaktfreien Transitionen in einem Schritt zu schalten. Man beachte, daß die Transitionen durch die Beschränkung auf eine Marke pro Stelle auch im Nachbereich disjunkt sein müssen.

Definition 4.20 [Aktivierung und Schalten von Schritten in elementaren

FSNets nach Referenzsemantik]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet, m_1, m_2 Markierungen von $EFSN$ und $\mathrm{A} \subseteq \mathrm{T}$ eine kontaktfreie Transitionsmenge, also eine Transitionsmenge für die gilt: $\forall t, t' \in \mathrm{A}, t \neq t' : (\bullet t \cup t\bullet) \cap (\bullet t' \cup t'\bullet) = \emptyset$. Die Wirkung von A sei definiert als

$$\mathrm{w}(\mathrm{A}) := \bigsqcup\{\mathrm{r_a}(t) \mid t \in \mathrm{A}\} \sqcup \mathrm{copymark}(\mathrm{S} - \bullet\mathrm{A} - \mathrm{A}\bullet).$$

(a) A heißt aktiviert (activated, firable), falls:

$$\mathrm{w}(\mathrm{A}) \sqcup \mathsf{pre}{::}\mathrm{m}_1 \text{ ist definiert}$$

(b) A schaltet m_1 zu m_2, symbolisch $\mathrm{m}_1 \xrightarrow{\mathrm{A}} \mathrm{m}_2$ (fires m_1 to m_2), falls A in m_1 aktiviert ist und:

$$\mathrm{m}_2 \sim (\mathrm{w}(\mathrm{A}) \sqcup \mathsf{pre}{::}\mathrm{m}_1)@\mathsf{post} \qquad \Diamond$$

Die Wirkung einer Menge von Transitionen ergibt sich also als die Unifikation ihrer abgeleiteten Transitionsregeln mit nachträglicher Anwendung der copymark-Funktion. Durch die Forderung, daß die Transitionen kontaktfrei sind, kann die Unifikation nicht scheitern.

Die Definition einer Schrittsemantik nach Wertsemantik ergibt sich analog zur Schaltfolgensemantik und wird hier nicht wiedergegeben.

Wiederum soll die Konstruktion der Wirkung eines Schrittes an dem bekannten Beispiel veranschaulicht werden. Im Beipiel aus Abbildung 4.10 sind t2 und t4 kontaktfreie Transitionen. Abbildung 4.22 zeigt die Konstruktion und das Ergebnis der Wirkung des Schritts $\{\mathrm{t2}, \mathrm{t4}\}$. Man kann deutlich erkennen, daß alle an der Unifikation beteiligten Terme zwar dieselben Features pre und post besitzen, diese aber disjunkte Unter-Features aufweisen, die sich in beiden Fällen genau zu der Menge aller Stellen ergänzen.

4.2.7 Erreichbare Markierungen in elementaren FSNets

Für Basis-FSNets wurde in Abschnitt 4.1.3 eine Schaltfolgensemantik definiert, aus der sich analog zu den klassischen Methoden der Petrinetztheorie die Menge der erreichbaren Markierungen eines Netzes ergibt. Ein Erreichbarkeitsgraph (siehe Abschnitt 3.2.2) stellt diese Menge und die Schaltfolgen, die zu den jeweiligen Markierungen führen, übersichtlich dar.

Für elementare FSNets soll das Thema hier noch einmal aufgegriffen werden, da sich durch die Vereinfachungen gegenüber Basis-FSNets auch der Erreichbarkeitsgraph vereinfachen läßt. An den Kanten muß nur die Transition, nicht ein zusätzlicher Schaltmodus angegeben werden, da in elementaren FSNets eine Transition immer nur in einem Schaltmodus zur Zeit aktiviert sein kann.

$$\mathrm{w}(\{\mathrm{t2},\mathrm{t4}\}) = \mathrm{r_a}(\mathrm{t2}) \sqcup \mathrm{r_a}(\mathrm{t4}) \sqcup \mathrm{copymark}(\{\mathrm{p1},\mathrm{p3}\})$$

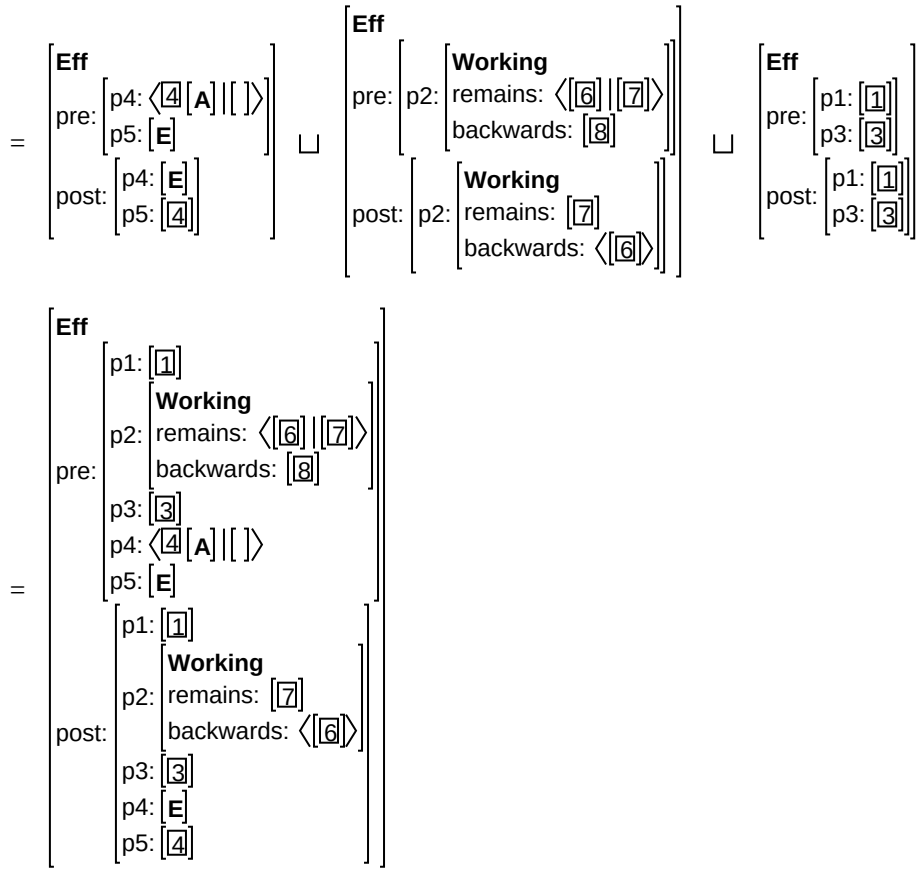

Abbildung 4.22: Die Wirkung des Schritts $\{\mathrm{t2},\mathrm{t4}\}$ aus dem Netz aus Abbildung 4.10.

Da die Menge der erreichbaren Markierungen eines elementaren FSNet im Gegensatz zu B/E-Netzen nicht endlich sein muß, gilt dies ebenso für die Anzahl der Knoten des Erreichbarkeitsgraphen. Ob es eine zum Überdeckungsgraphen bei S/T-Netzen analoge Konstruktion bei EFSNets gibt, stellt eine interessante Fragestellung dar, der hier nicht weiter nachgegangen wird.

Je nachdem, welche Schaltsemantik angewandt wird, erhält man unterschiedliche Erreichbarkeitsgraphen, die sich jedoch in ihrer Struktur sehr ähnlich sind.

Abbildung 4.23 zeigt den Erreichbarkeitsgraphen des Netzes aus Abbildung 4.10, wobei die in Abschnitt 2.3.1 eingeführte Mischnotation aus Graph- und AVM-Notation für die Feature Structures genutzt wird. Es wird wie zuvor eine Referenzsemantik angenommen. Die Knoten des Erreichbarkeitsgraphen sind mit RGN (*reachability graph node*) bezeichnet. Zum Vergleich zeigt Abbildung 4.23 den Erreichbarkeitsgraphen für die Wertsemantik-Version, die in Abbildung 4.11 dargestellt

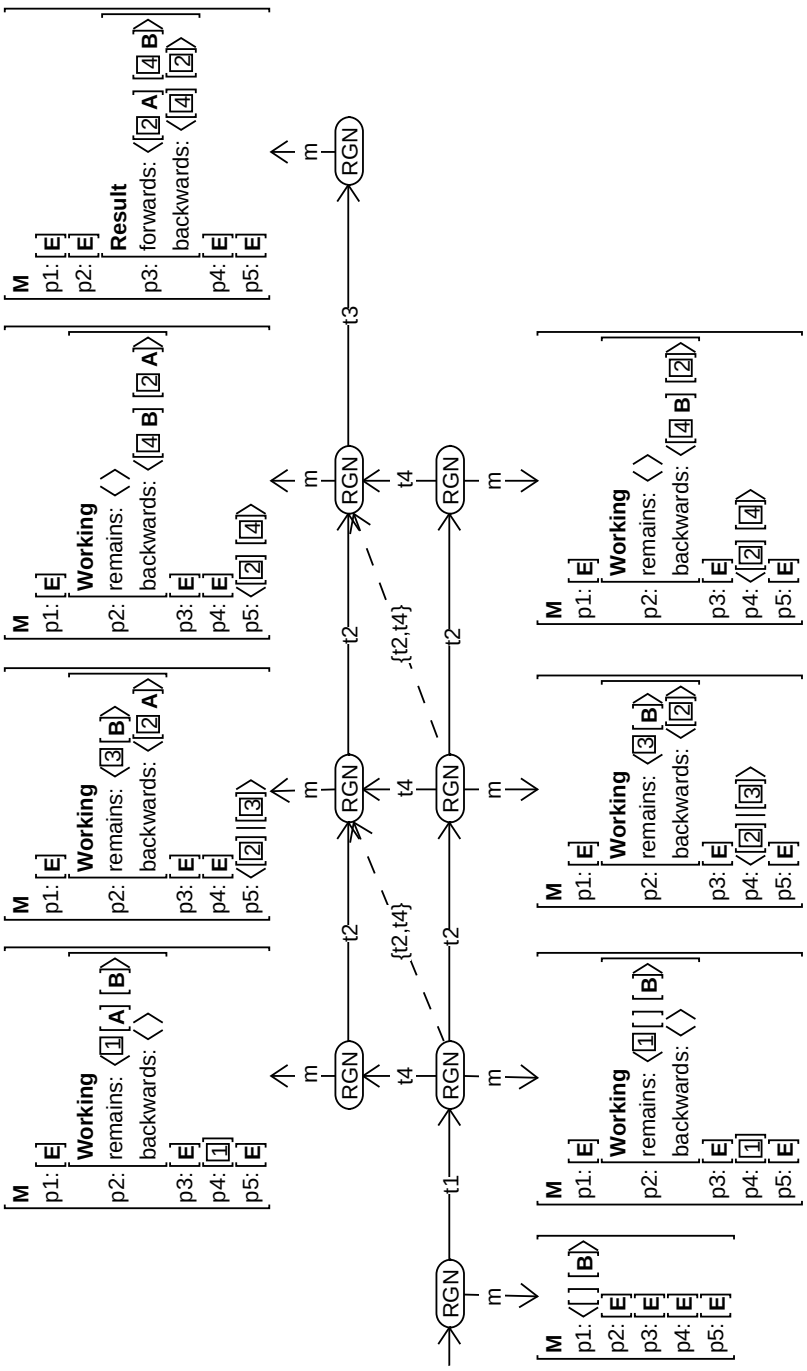

Abbildung 4.23: Der Erreichbarkeitsgraph des Netzes aus Abbildung 4.10 nach Referenzsemantik.

wurde. Aus den Erreichbarkeitsgraphen lassen sich die drei möglichen Schaltfolgen ablesen, die in beiden Varianten die gleichen sind: t1 t2 t2 t4 t3 sowie t1 t2 t4 t2 t3 und t1 t4 t2 t2 t3.

Die Unterschiede werden daran deutlich, wie die Information, daß das erste Element der Liste vom Typ A ist, propagiert wird. In der Referenzsemantik kann man erkennen, wie durch die Knotenreferenzen [2] und [4] auf die beiden Elemente der Liste aus Marken in verschiedenen Stellen verwiesen wird. Es handelt sich also nicht um eine Wertmarkierung nach Definition 4.10. Sobald t4 geschaltet hat, also sobald ein Zustand im oberen Bereich des Erreichbarkeitsgraphen in Abbildung 4.23 eingenommen wird, ist die Information durch die Koreferenz im gesamten Netz bekannt. In der Wertsemantik wird die Information erst beim Schalten von t3 weiter propagiert.

Wie die Abbildungen nahelegen, ist der Erreichbarkeitsgraph eines elementaren FSNets wiederum eine Feature Structure. Auf eine formale Definition des Erreichbarkeitsgraphen als Feature Structure soll hier aber verzichtet werden.

Betrachtet man die erste Erweiterung der Schaltsemantik, die im vorherigen Abschnitt vorgestellte Schrittsemantik, so stellt man fest, daß im Erreichbarkeitsgraphen zusätzliche Kanten entstehen, die nach Definition 4.20 kontaktfreie Transitionen gleichzeitig schalten lassen. In den Abbildungen sind diese Kanten gestrichelt dargestellt und mit der (Multi-)Menge der schaltenden Transitionen beschriftet. Die Kanten der Schaltfolgensemantik können als einelementige Multimengen interpretiert werden und stellen somit auch (sehr einfache) Schritte dar. Im Beispiel gibt es nur zwei Kanten, bei denen die Menge mehr als ein Element enthält, da nur die Transitionen t2 und t4 kontaktfrei sind.

Die dritte operationale Semantik, die für elementaren FSNets definiert werden soll, ist eine Prozeßsemantik. Diese wird im nächsten Abschnitt eingeführt, wobei wiederum Wert- und Referenzsemantik berücksichtigt werden.

4.2.8 Eine Prozeßsemantik für elementare FSNets

Nachdem für elementare FSNets nun eine Schaltfolgen- und eine Schrittsemantik definiert wurden, soll in diesem Abschnitt eine Prozeßsemantik angegeben werden. In einer Prozeßsemantik werden, wie in Abschnitt 3.2 dargestellt, nicht sequentielle Schaltfolgen betrachtet, sondern ein Ablauf wird wiederum durch ein Petrinetz beschrieben.

Abläufe von elementaren FSNets haben den Vorteil, daß in diesen zwar Nebenläufigkeit vorkommen kann, durch die Beschränkung auf höchstens eine Marke pro Stelle aber relativ einfache Prozeßstrukturen entstehen. Dadurch ist es möglich, Prozesse von EFSNets als Feature Structures darzustellen. Dies hat nicht nur den Vorteil, für die Konstruktion von Prozessen einen einheitlichen Formalismus nutzen und auf die Ergebnisse aus Abschnitt 2.3 zurückgreifen zu können. In Abschnitt 4.3 werden darüber hinaus die Marken in FSNets als Prozesse aufgefaßt, um Netze in

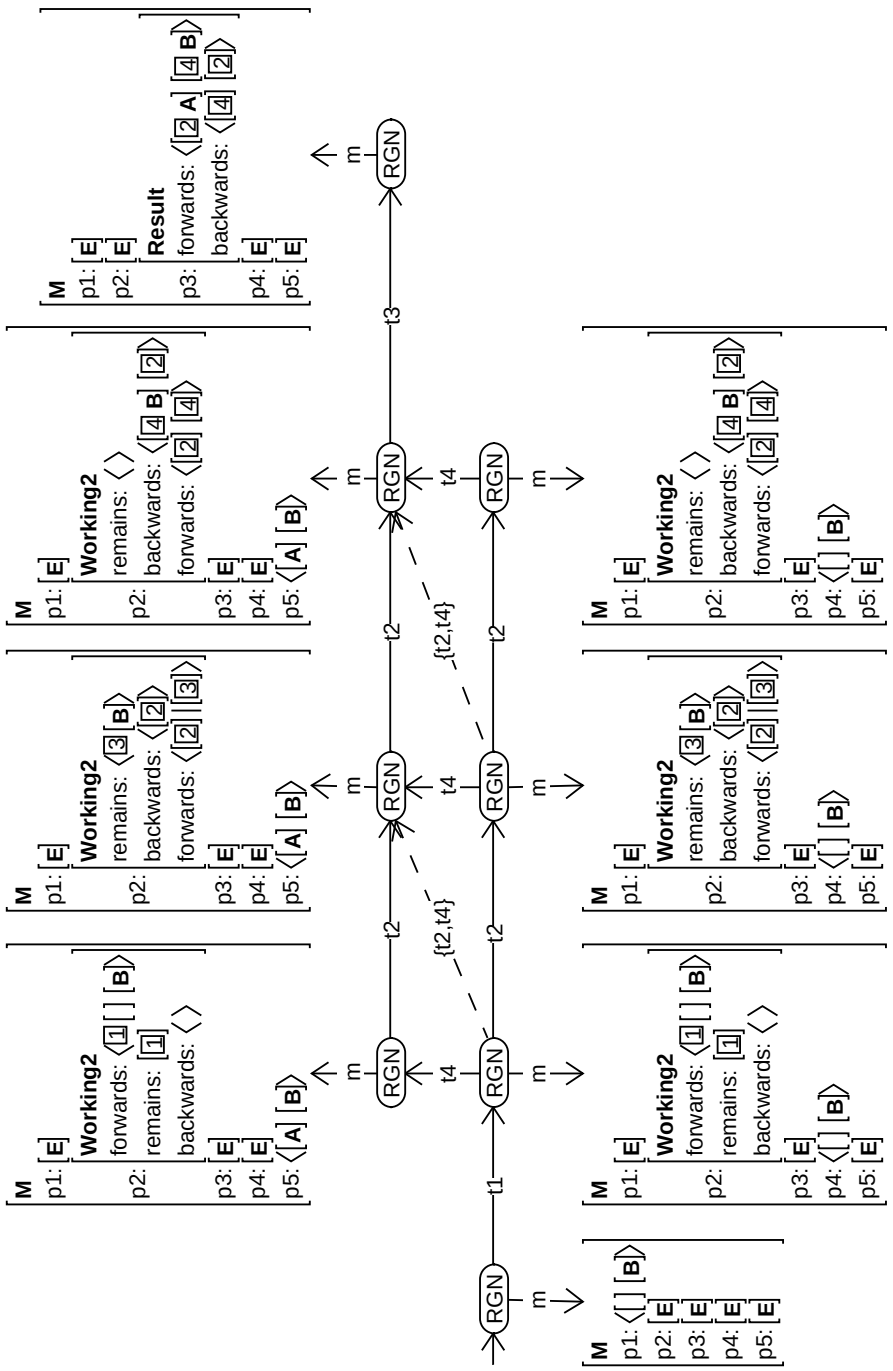

Abbildung 4.24: Der Erreichbarkeitsgraph des Netzes aus Abbildung 4.11 nach Wertsemantik.

Netzen als FSNets darzustellen. Die in Abschnitt 3.3.1 vorgestellte Unifikation von Prozessen läßt sich durch Unifikation von Feature Structures darstellen.

In den folgenden Unterabschnitten werden zunächst Prozesse von B/E-Netzen und dann von EFSNets als Feature Structures dargestellt, was uns zu einer Prozeßsemantik für EFSNets führt.

Prozesse als Feature Structures

Prozesse von Petrinetzen werden wiederum als Petrinetze dargestellt. Eine besondere Eigenschaft solcher Prozeßnetze ist, daß sie keine Konflikte enthalten, also Kausalnetze darstellen. Eine Abbildung eines Kausalnetzes in ein Auftragssystem ist immer möglich (siehe Abschnitt 3.2.1). Aus der Definition des Kausalnetzes folgt, daß eine Stelle eines Prozeßnetzes höchstens eine Eingangs- und eine Ausgangstransition besitzt. Die Stellen eines Kausalnetzes stehen für die Kanten des Auftragssystems, welche die direkten Präzedenzen zwischen den Aufträgen darstellen.

Wir wollen nun Prozeßnetze als Feature Structures darstellen, wobei wir zunächst Prozesse von B/E-Netzen (siehe Abschnitt 3.2.1, Definition B.7) betrachten. Im Prinzip wäre es möglich, für Stellen und Transitionen jeweils einen Typ einzuführen, durch den die beiden Knotenarten des Petrinetzes im Feature-Structure-Graph unterschieden werden. Es gibt aber mehrere Gründe, die dagegen sprechen, die Darstellung so zu wählen:

(a) Die Kanten eines B/E-Netz-Prozesses sind unbenannt (bzw. lediglich mit der Kantenbewertung 1 beschriftet), so daß für die Features der Feature-Structure-Darstellung keine sinntragenden Namen vergeben werden könnten.

(b) Es soll eine Darstellung gewählt werden, die das Unifikationsverhalten von Prozessen adäquat auf Feature Structures überträgt. Da bei der Unifikation von Feature Structures die Features bestimmen, welche Knoten identifiziert werden, müssen die Features die relevante Struktur des Prozesses widerspiegeln.

(c) Die oben beschriebene einfachere Struktur des Prozeßnetzes würde nicht ausgenutzt. Es würde sich eine komplexere Struktur ergeben, die nachträglich durch zusätzliche Einschränkungen wieder so eingeschränkt werden muß, daß in der Feature-Structure-Darstellung keine Konflikte auftreten können.

Die folgende Definition eines B/E-Netz-Prozesses als Feature Structure modelliert deshalb nicht die Struktur des Kausalnetzes, sondern die des zugeordneten Auftragssystems. Aufträge, die den Transitionen des Prozesses entsprechen, werden in der Feature Structure als Knoten dargestellt. Kanten des Auftragssystems, die aus Stellen des Kausalnetzes resultieren, werden zu Kanten der Feature Structure, die hier im Gegensatz zum Auftragssystem benannt sind. Dabei ist zu beachten, daß Stellen des Kausalnetzes nur dann in der Feature Structure dargestellt werden,

wenn sie im *Vor*bereich einer Transition auftreten, da – wie beim Auftragssystem – nur Präzedenzen zwischen Transitionen betrachtet werden.

Ein letzter Aspekt, der Feature Structures und Kausalnetze bzw. Auftragssysteme unterscheidet, ist, daß Feature Structures einen Wurzelknoten besitzen müssen. Hier liegt es nahe, eine genau einmal zu Beginn des Prozesses auftretende Initialisierungstransition hinzuzufügen, welche die Anfangsmarkierung des unterliegenden Netzes erzeugt.

Wir definieren zunächst ein spezielles Prozeß-Typsystem, das Typen für Transitionen und Features für Stellen des unterliegenden B/E-Netzes bereitstellt. Ein zusätzlicher Typ Proc dient einerseits als gemeinsamer Obertyp aller Transitionen, andererseits als Typ für die Initialisierungstransition.

Definition 4.21 [B/E-Prozeß-Typsystem]
Sei $BEN = \langle \mathrm{S}, \mathrm{T}, \mathrm{F}, \mathrm{m}_0 \rangle$ ein B/E-Netz. Das *B/E-Prozeß-Typsystem* BTS_{BEN} zu BEN ergibt sich als
$\mathrm{BTS}_{BEN} := \langle \mathrm{Type}_\mathrm{B}, (\sqsubseteq_\mathrm{B}), \mathrm{Feat}_\mathrm{B}, (\mathrm{approp}_\mathrm{B}) \rangle$ mit:

(a) $\mathrm{Type}_\mathrm{B} := \{\mathsf{Proc}\} \cup \mathrm{T}$, wobei o.B.d.A. gelte $\mathsf{Proc} \notin \mathrm{T}$,

(b) $\sqsubseteq_\mathrm{B}$ sei die reflexive Hülle über das kartesische Produkt $\{\mathsf{Proc}\} \times \mathrm{T}$,

(c) $\mathrm{Feat}_\mathrm{B} := \mathrm{S}$ und

(d) $\forall s \in \mathrm{S}, t \in \mathrm{T} : \mathrm{approp}_\mathrm{B}(\mathsf{Proc}, s) := \mathsf{Proc} \quad \wedge \quad \mathrm{approp}_\mathrm{B}(t, s) := \mathsf{Proc}$, ansonsten sei $\mathrm{approp}_\mathrm{B}$ undefiniert. ◊

Es ließen sich noch speziellere Typen definieren, die z.B. für eine Transition nur die Stellen als Features erlauben, auf welche diese Transition Marken ablegt. Um das Prozeß-Typsystem übersichtlich zu halten, wurde auf solch eine strengere Typisierung verzichtet, da sich ein wohlgeformter Prozeß ohnehin nur mit zusätzlichen Bedingungen definieren läßt, die sich nicht durch Typisierung ausdrücken lassen.

In der folgenden Definition eines B/E-Netz-Prozesses als Feature Structure wird gefordert, daß das B/E-Netz einem 1-sicheren S/T-Netz entspricht, damit keine Konflikte im Nachbereich auftreten können. Diese Eigenschaft kann für jedes S/T-Netz, das nach Definition B.7 ein B/E-Netz ist, durch die in Abschnitt 3.2.2 erwähnte Konstruktion von Komplementärstellen sichergestellt werden.

Definition 4.22 [B/E-Netz-Prozeß als Feature Structure]
Sei $BEN = \langle \mathrm{S}, \mathrm{T}, \mathrm{F}, \mathrm{m}_0 \rangle$ ein B/E-Netz, das einem 1-sicheren S/T-Netz entspricht und BTS_{BEN} das B/E-Prozeß-Typsystem zu BEN. Eine wohltypisierte Feature Structure $BEP \in \mathrm{WFS}_{\mathrm{BTS}_{BEN}}$ heißt ein *B/E-Prozeß* von BEN, wenn die folgenden Bedingungen erfüllt sind:

(a) $\forall \pi, \pi' \in \mathrm{S}^* : BEP@\pi = BEP@\pi\,\pi' \Rightarrow \pi' = \varepsilon$
(die Prozeßstruktur ist azyklisch),

(b) $\theta(BEP) = \mathsf{Proc}$
(der Wurzelknoten entspricht der Initialisierungstransition),

(c) $\forall \pi \in \mathrm{S}^+ : BEP@\pi$ ist definiert $\Rightarrow \theta(BEP@\pi) \in \mathrm{T}$
(alle anderen Knoten entsprechen „echten“ Transitionen),

(d) $\forall s \in \mathrm{S} : BEP@s$ ist definiert $\Rightarrow \mathrm{m}_0(s) > 0$
(die Initialisierungstransition erzeugt nur Marken der Anfangsmarkierung),

(e) $\forall \pi \in \mathrm{S}^+, t \in \mathrm{T}, s \in \mathrm{S} : t = \theta(BEP@\pi) \Rightarrow (s \in {\bullet t} \Leftrightarrow \exists \pi' : BEP@\pi'\, s = BEP@\pi)$
(jede Transition konsumiert alle Marken ihres Vorbereichs),

(f) $\forall \pi \in \mathrm{S}^+, t \in \mathrm{T}, s' \in \mathrm{S} : t = \theta(BEP@\pi) \wedge BEP@\pi\, s'$ ist definiert $\Rightarrow s' \in t{\bullet}$
(jede Transition erzeugt nur Marken ihres Nachbereichs). $\Diamond$

B/E-Netz Prozesse sind somit als Feature Structures definiert, wobei deutlich geworden ist, wie die Abbildung von Kausalnetzen bzw. Auftragssystemen auf Feature Structures vorgenommen wird. Diese einfachere Abbildung ist hilfreich, wenn im nächsten Abschnitt Prozesse elementarer FSNets definiert werden, die als gefärbte Prozesse eine höhere Komplexität aufweisen.

Definition einer Prozeßsemantik elementarer FSNets

Elementare FSNets können als eine Art gefärbte B/E-Netze betrachtet werden. Wie bei B/E-Netzen kann jede Stelle nur höchstens eine Marke enthalten, allerdings weisen die Marken in elementaren FSNets eine Struktur auf. Dadurch werden wie bereits in Abschnitt 4.2.7 erwähnt im Gegensatz zu B/E-Netzen unendliche Erreichbarkeitsgraphen möglich.

In diesem Abschnitt wird die Darstellung von Prozessen als Feature Structures von B/E-Netz-Prozessen auf Prozesse elementarer FSNets, sogenannte *EFSNet-Prozesse*, erweitert. Bezüglich der Prozesse erfordern elementare FSNets gegenüber B/E-Netzen eine logische Einschränkung. Die Prozesse des unterliegenden B/E-Netzes sind Kandidaten für die gefärbten Prozesse, in denen zusätzlich die Transitionsregeln zwischen den beim Vorkommen einer Transition konsumierten und erzeugten Marken gelten müssen.

Die Features dienen im B/E-Prozeß-Typsystem dazu, eine kausale Abhängigkeit zwischen zwei Transitionsvorkommen mit dem Namen der entsprechenden Stelle des unterliegenden Netzes zu beschriften. Für jede dieser Abhängigkeiten muß nun eine Feature Structure als Marke angegeben werden können. Dazu gibt es prinzipiell zwei Möglichkeiten: Wenn die Stellen des Prozesses durch Knoten in der Feature Structure dargestellt würden, könnten diese die Marke sowie ihren optionalen Konsumenten beschreiben. Diese Variante wurde aber aus den oben genannten Gründen nicht gewählt. Stattdessen wird die Struktur des B/E-Netz-Prozesses übernommen

und für Marken erweitert. Auch hier gibt es wiederum zwei Möglichkeiten: Man könnte für jede Stelle zusätzlich zum Konsumenten-Feature ein weiteres Feature direkt in den Transitionsknoten aufnehmen. Um eine übersichtlichere Struktur zu erhalten und die Namen der Stellen wiederverwenden zu können, wird hier die zweite Möglichkeit gewählt: Jeder Knoten, der ein Transitionsvorkommen beschreibt, erhält *ein* zusätzliches Feature m, das eine Teilmarkierung als Wert hat, die für alle von dieser Transition veränderten Stellen den neuen Wert beschreibt.

Wir definieren zunächst wiederum ein spezielles Prozeß-Typsystem, das neben den Stellen und Transitionen nun auch Feature-Structure-Marken in der soeben beschriebenen Weise darstellen kann. Dabei wird auf das zuvor definierte Transitionsregel-Typsystem aufgebaut, da auch für Prozesse eine spezielle Darstellung der Transitionsregeln abgeleitet werden soll. Wie beim Erstellen abgeleiteter Transitionsregeln gibt es auch hier einen Hilfstyp (PEff), der nur während der Konstruktion benötigt wird, im Prozeß selbst aber nicht mehr auftritt.

Definition 4.23 [Prozeß-Typsystem]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet, $\mathrm{RTS}_{EFSN} = \langle \mathrm{Type}_\mathrm{R}, \sqsubseteq_\mathrm{R}, \mathrm{Feat}_\mathrm{R}, \mathrm{approp}_\mathrm{R} \rangle$ das sich nach Definition 4.12 aus $EFSN$ ergebende Transitionsregel-Typsystem und $\top_\mathrm{M}$ der allgemeinste Typ aus RTS_{EFSN}. Das *Prozeß-Typsystem* PTS_{EFSN} zu $EFSN$ sei definiert als
$\mathrm{PTS}_{EFSN} := \langle \mathrm{Type}_\mathrm{R} \cup \mathrm{Type}_\mathrm{P}, (\sqsubseteq_\mathrm{P}), \mathrm{Feat}_\mathrm{R} \cup \mathrm{Feat}_\mathrm{P}, (\mathrm{approp}_\mathrm{R}) \cup (\mathrm{approp}_\mathrm{P}) \rangle$ mit:

(a) $\mathrm{Type}_\mathrm{P} := \{\mathsf{Proc}, \mathsf{PEff}\} \cup \mathrm{T}$,

(b) $\sqsubseteq_\mathrm{proc}$ ist die transitive, reflexive Hülle über $(\sqsubseteq_\mathrm{R}) \cup \{(\top_\mathrm{M}, \mathsf{Proc})\} \cup (\{\mathsf{Proc}\} \times \mathrm{T}) \cup \{(\mathsf{Eff}, \mathsf{PEff})\}$,

(c) $\mathrm{Feat}_\mathrm{P} := \{\mathsf{m}, \mathsf{postc}, \mathsf{proc}\}$ und

(d) $\mathrm{approp}_\mathrm{P}(\mathsf{PEff}, \mathsf{pre}) := \mathsf{M}$ und
$\mathrm{approp}_\mathrm{P}(\mathsf{PEff}, \mathsf{post}) := \mathsf{M}$ (geerbt),
$\mathrm{approp}_\mathrm{P}(\mathsf{PEff}, \mathsf{postc}) := \mathsf{M}$,
$\mathrm{approp}_\mathrm{P}(\mathsf{PEff}, \mathsf{proc}) := \mathsf{Proc}$,
$\mathrm{approp}_\mathrm{P}(\mathsf{Proc}, \mathsf{m}) := \mathsf{M}$ und
$\forall s \in \mathrm{S}, t \in \mathrm{T} : \mathrm{approp}_\mathrm{P}(\mathsf{Proc}, s) = \mathsf{Proc}$, $\mathrm{approp}_\mathrm{P}(t, s) = \mathsf{Proc}$ (geerbt), und $\mathrm{approp}_\mathrm{P}(\mathsf{PEff}, s) = \mathsf{Proc}$,
ansonsten sei $\mathrm{approp}_\mathrm{P}$ undefiniert,

wobei o.B.d.A. gelte $\mathrm{Type}_\mathrm{R} \cap \mathrm{Type}_\mathrm{P} = \mathrm{Feat}_\mathrm{R} \cap \mathrm{Feat}_\mathrm{P} = \emptyset$. ◊

Die Bedeutung des Typs PEff wird bei der Konstruktion der Prozeß-Transitionsregeln deutlich werden. Man kann diese Datenstruktur zum einen als Effekt einer Transition auf den Prozeß ansehen, zum anderen als Beschreibung eines S-Schnitts des Prozesses (siehe Kapitel 3.2.2). Die Features von PEff sind ein Prozeß (Feature proc vom Typ Proc), die freien Anschlußstellen im Prozeß für jede

Stelle (Features aus S vom Typ Proc) und die von Eff geerbten Features pre und post sowie ein weiteres Feature postc vom Markierungs-Typ M. Durch pre und post wird mit Hilfe der abgeleiteten Transitionsregel aus der aktuellen Markierung die Folgemarkierung erzeugt. postc ist entweder identisch mit post (Referenzsemantik) oder enthält Kopien der Marken unter post (Wertsemantik).

Wie zuvor soll die Konstruktion des Prozeß-Typsystems anhand eines Beispiels veranschaulicht werden. Dazu zeigt Abbildung 4.25 das Prozeß-Typsystem zu dem Netz aus Abbildung 4.10.

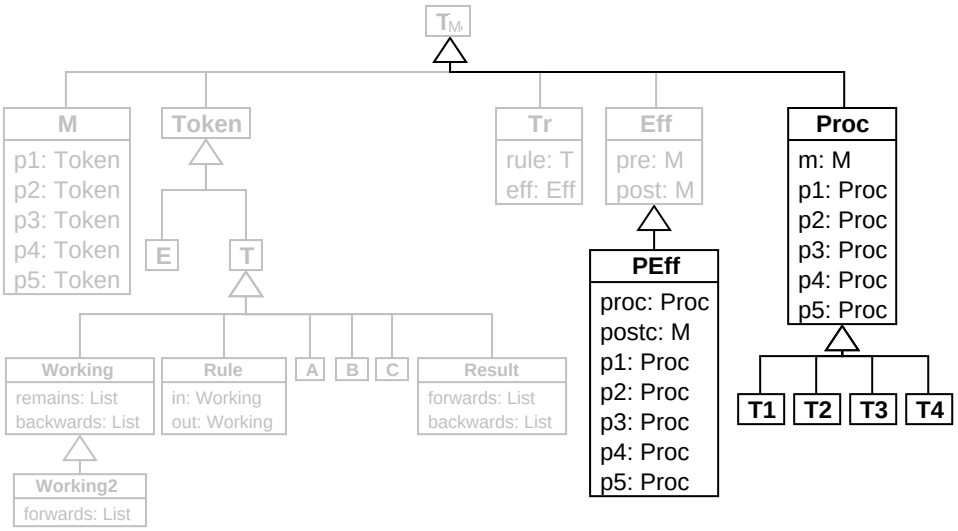

Abbildung 4.25: Das Prozeß-Typsystem zu dem Netz aus Abbildung 4.10.

Basierend auf diesem Typsystem wird nun eine grundlegende EFSNet-Prozeßstruktur definiert, in der noch nicht die Markenwerte betrachtet werden. Die Unterschiede zur Definition des B/E-Netz-Prozesses kommen durch die Färbung des Netzes und die Behandlung leerer Stellen zustande, da hier nicht mehr gefordert wird, daß das Netz 1-sicher ist.

Definition 4.24 [EFSNet-Prozeßstruktur]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet und PTS_{EFSN} das Prozeß-Typsystem zu $EFSN$. Eine wohltypisierte Feature Structure $P \in \mathrm{WFS}_{\mathrm{PTS}_{EFSN}}$ heißt eine *EFSNet-Prozeßstruktur* von $EFSN$, wenn die folgenden Bedingungen erfüllt sind:

(a) $\forall \pi, \pi' \in \mathrm{S}^* : P@\pi = P@\pi\,\pi' \Rightarrow \pi' = \varepsilon$
(die kausale Prozeßstruktur ist azyklisch),

(b) $\theta(P) = \mathsf{Proc}$
(der Wurzelknoten entspricht der Initialisierungstransition),

(c) $\forall \pi \in \mathrm{S}^+ : P@\pi$ ist definiert $\Rightarrow \theta(P@\pi) \in \mathrm{T}$
(alle anderen Prozeß-Knoten entsprechen „echten“ Transitionen),

(d) $\forall \pi \in \mathrm{S}^+, t \in \mathrm{T}, s \in \mathrm{S} : t = \theta(P@\pi) \Rightarrow (s \in {\bullet t} \cup t{\bullet} \Leftrightarrow \exists \pi' : P@\pi'\, s = P@\pi)$
(jede Transition konsumiert alle Marken ihres Wirkungsbereichs),

(e) $\forall \pi \in \mathrm{S}^+, t \in \mathrm{T}, s \in \mathrm{S} : t = \theta(P@\pi) \Rightarrow$
$(P@\pi\, \mathsf{m}\, s$ ist definiert $\Leftrightarrow s \in {\bullet t} \cup t{\bullet} \ \wedge\ P@\pi\, s$ ist definiert $\Rightarrow s \in {\bullet t} \cup t{\bullet})$
(jede Transition erzeugt Marken für ihren gesamten Wirkungsbereich und nur diese können konsumiert werden). $\Diamond$

Um zu korrekt markierten EFSNet-Prozessen zu gelangen, muß jedes Transitionsvorkommen im Prozeß die entsprechende Transitionsregel erfüllen. Dazu definieren wir die Vorbereichsmarkierung einer Transition in einem EFSNet-Prozeß.

Definition 4.25 [Vorbereichsmarkierung]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet und PTS_{EFSN} das Prozeß-Typsystem zu $EFSN$. Die *Vorbereichsmarkierungs-Funktion* $\circ : \mathrm{WFS}_{\mathrm{PTS}_{EFSN}} \times \mathrm{S}^+ \hookrightarrow \mathrm{WFS}_{\mathrm{PTS}_{EFSN}}$ liefert zu einem EFSNet-Prozeß P von $EFSN$ und einem Pfad $\pi \in \mathrm{S}^+$, der auf eine der Transitionen des Prozesses zeigt ($P@\pi$ ist definiert), die allgemeinste Feature Structure $\circ(P, \pi) = P \circ \pi := PRE$, für die gilt:

- $\theta(PRE) = \mathsf{PEff}$,
- $PRE@\mathsf{proc} = P$,
- $\forall s \in \mathrm{S}, \pi' \in \mathrm{S}^* : P@\pi'\, s = P@\pi \Rightarrow PRE@\mathsf{pre}\, s = PRE@\mathsf{proc}\, \pi'\, \mathsf{m}\, s$. $\Diamond$

Ein EFSNet-Prozeß ist bezüglich Wertsemantik genau dann wohlgeformt, wenn die initiale Transition genau die Anfangsmarkierung erzeugt und sich für jedes Transitionsvorkommen aus der Anwendung der abgeleiteten Transitionsregel auf die Vorbereichsmarkierung genau die vom Transitionsvorkommen erzeugte Nachbereichsmarkierung ergibt.

Definition 4.26 [Wertsemantik-EFSNet-Prozeß]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet und PTS_{EFSN} das Prozeß-Typsystem zu $EFSN$. Eine Feature Structure $P \in \mathrm{WFS}_{\mathrm{PTS}_{EFSN}}$ heißt ein *nach Wertsemantik wohlgeformter EFSNet-Prozeß* von $EFSN$, wenn P ein EFSNet-Prozeß von $EFSN$ ist und gleichzeitig die allgemeinste Feature Structure darstellt, die folgende zusätzliche Bedingungen erfüllt:

(a) $\mathrm{m}_0 \sim P@\mathsf{m}$
(Anfangsmarkierung und von der Initialisierungstransition erzeugte Markierung sind gleich),

(b) $\forall \pi \in \mathrm{S}^+, t \in \mathrm{T}:$
$t = \theta(\pi) \Rightarrow \mathrm{valmark}((P \circ \pi \sqcup \mathrm{r_a}(t))@\mathsf{post}) \sim P@\pi\,\mathsf{m}$
(die Kopie der Markierung, die sich aus der Anwendung der abgeleiteten Transitionsregel auf die Vorgängermarkierung ergibt, gleicht der vom Transitionsvorkommen erzeugten Folgemarkierung). $\Diamond$

In einem Referenzsemantik-Prozeß dürfen hingegen Marken durch verschiedene Transitionsvorkommen spezialisiert werden. Deswegen erlauben wir, daß die Marken im Prozeß spezieller sein können, als von der Anfangsmarkierung oder einer Transitionsregel gefordert. Da wie zuvor die *allgemeinste* Feature Structure gewählt wird, welche die Bedingungen erfüllt, kann trotzdem keine Information hinzugefügt werden, die nicht in der Anfangsmarkierung oder den Transitionsregeln enthalten ist.

Definition 4.27 [Referenzsemantik-EFSNet-Prozeß]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet und PTS_{EFSN} das Prozeß-Typsystem zu $EFSN$. Eine Feature Structure $P \in \mathrm{WFS}_{\mathrm{PTS}_{EFSN}}$ heißt ein *nach Referenzsemantik wohlgeformter EFSNet-Prozeß* von $EFSN$, wenn P ein EFSNet-Prozeß von $EFSN$ ist und gleichzeitig die allgemeinste Feature Structure darstellt, die folgende zusätzliche Bedingungen erfüllt:

(a) $\mathrm{m}_0 \sqsubseteq P@\mathsf{m}$
(die Anfangsmarkierung subsumiert die von der Initialisierungstransition erzeugte Markierung),

(b) $\forall \pi \in \mathrm{S}^+, t \in \mathrm{T}:$
$t = \theta(\pi) \Rightarrow (P \circ \pi \sqcup \mathrm{r_a}(t))@\mathsf{post} \sqsubseteq P@\pi\,\mathsf{m}\,s$
(die Markierung, die sich aus der Anwendung der abgeleiteten Transitionsregel auf die Vorgängermarkierung ergibt, subsumiert die vom Transitionsvorkommen erzeugte Folgemarkierung). $\Diamond$

Die Bedingung in der Referenzsemantik, daß die Anwendung der Transitionsregel die Folgemarkierung subsumiert, führt bei Koreferenzen auf Marken zu Unifikation. Da Koreferenzen auf Marken dazu führen, daß von einer Substruktur gefordert wird, daß sie *von mehreren* Feature Structures subsumiert wird und nach Definition die allgemeinste Feature Structure zu wählen ist, welche die Bedingungen erfüllt, erhält man das Ergebnis gerade durch Unifikation.

Wenden wir uns nun der Konstruktion von wohlgeformten EFSNet-Prozessen elementarer FSNets zu. Dazu definieren wir zunächst den maximalen S-Schnitt eines EFSNet-Prozesses, aus dem sich die maximale Markierung des unterliegenden elementaren FSNets ergibt.

Definition 4.28 [Maximaler S-Schnitt eines EFSNet-Prozesses]
Sei $EFSN$ ein elementares FSNet und P ein EFSNet-Prozeß zu $EFSN$. Ein *maximaler S-Schnitt* von P, erzeugt durch die Funktion $\mathrm{scut} : \mathrm{WFS}_{\mathrm{PTS}_{EFSN}} \hookrightarrow \mathrm{WFS}_{\mathrm{PTS}_{EFSN}}$, ist die allgemeinste Feature Structure $\mathrm{scut}(P)$, für die gilt:

- $\theta(\text{scut}(P)) = \mathsf{PEff}$,
- $\text{scut}(P)@\mathsf{proc} \sim P$,
- $\theta(\text{scut}(P)@\mathsf{pre}) = \mathsf{M}$ und
- $\forall s \in \mathrm{S}, \pi \in \mathrm{S}^* :\ (P@\pi\,\mathsf{m}\,s \text{ ist definiert } \wedge P@\pi\,s \text{ ist undefiniert}) \Rightarrow$ $\Diamond$
  $(\text{scut}(P)@s = \text{scut}(P)@\mathsf{proc}@\pi \wedge$
  $\text{scut}(P)@s\,\mathsf{m}\,s = \text{scut}(P)@\mathsf{pre}\,s)$.

Ein S-Schnitt ist demnach eine Feature Structure vom Typ PEff, die unter dem Feature proc einen EFSNet-Prozeß, unter je einem Feature pro Stelle einen Verweis in den Prozeß und unter pre eine Markierung enthält. Die Verweise in den Prozeß zeigen für jede Stelle auf das bezüglich der Präzedenzrelation letzte Transitionsvorkommen, das eine Markierung (auch die „leere“ Markierung [E]) für diese Stelle erzeugt hat. Die Markierung zeigt auf den Wert dieser Marke, den man von dem Transitionsknoten aus für eine Stelle s durch den relativen Pfad $\mathsf{m}\,s$ findet. Da in elementaren FSNets höchstens eine Marke pro Stelle erlaubt ist, ergibt sich zu einem Prozeß immer ein eindeutiger maximaler S-Schnitt.

Weil der maximale S-Schnitt bei der Konstruktion eine EFSNet-Prozesses benötigt wird, soll gezeigt werden, wie sich dieser effektiv konstruieren läßt. In Definition 4.28 wird die allgemeinste Feature Structure gesucht, die bestimmte Bedingungen erfüllt. Man erhält eine solche Feature Structure durch Unifikation der Feature Structures, welche die Bedingungen repräsentieren. Gleichheit von Substrukturen wird durch Pfadgleichungen hergestellt, während Knoten mit bestimmten Typen durch das Hinschreiben der entsprechenden Feature Structure erzeugt und mit dem Vorpfad-Operator unter einen bestimmten Pfad gestellt werden.

Definition 4.29 [Konstruktion des maximalen S-Schnitts]
Sei $EFSN$ ein elementares FSNet und P ein EFSNet-Prozeß zu $EFSN$. $\text{scut}(P)$ wird folgendermaßen konstruiert:

$$\text{scut}(P) := \bigsqcup \Bigg\{ \begin{bmatrix} \mathsf{PEff} \\ \mathsf{pre}\colon [\mathsf{M}] \end{bmatrix},\ \mathsf{proc}{::}P,\ \bigsqcup_{\mathrm{U}} \{ s \doteq \mathsf{proc}\,\pi \mid s \in \mathrm{S}, \pi \in \mathrm{S}^*, P@\pi\,\mathsf{m}\,s \text{ ist def. } \wedge\ P@\pi\,s \text{ ist undef.} \},\ \bigsqcup_{\mathrm{U}} \{ s\,\mathsf{m}\,s \doteq \mathsf{pre}\,s \mid s \in \mathrm{S} \} \Bigg\}$$

$\Diamond$

Die Konstruktion des maximalen S-Schnitts soll an einem Beispiel veranschaulicht werden. Abbildung 4.26 zeigt einen EFSNet-Prozeß des Netzes aus Abbildung 4.11 nach Referenzsemantik (grau) und den dazugehörigen maximalen S-Schnitt (schwarz). Wie der Prozeß Schritt für Schritt konstruiert werden kann, wird

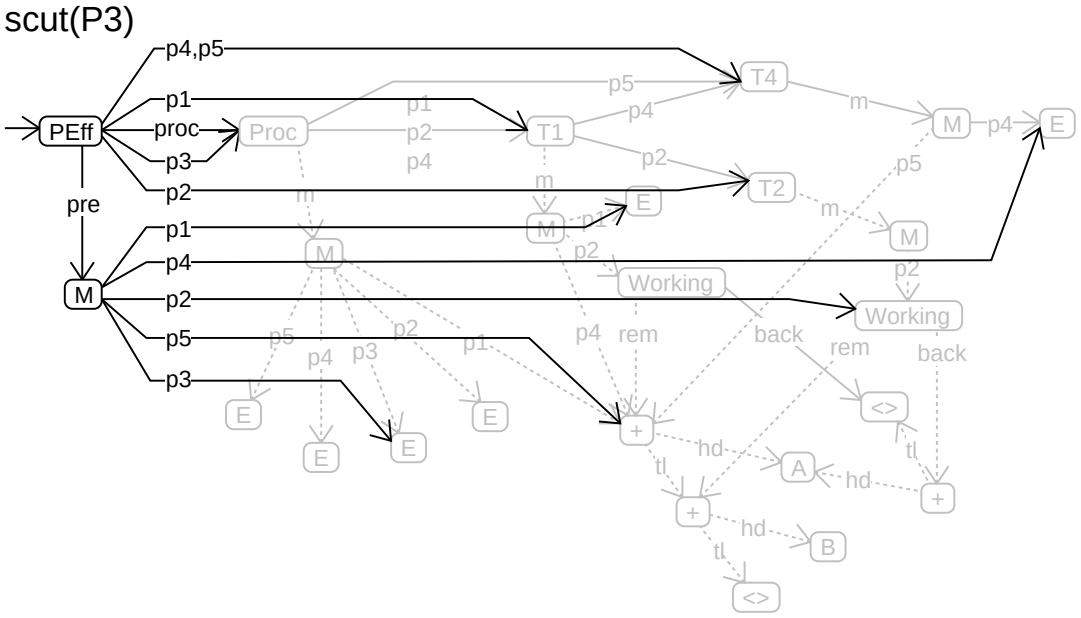

Abbildung 4.26: Ein EFSNet-Prozeß nach Referenzsemantik des Netzes aus Abbildung 4.11 (grau) und sein maximaler S-Schnitt (schwarz).

weiter unten beschrieben. Dort ist auch der gleiche S-Schnitt in AVM-Notation dargestellt (Abbildung C.5). Zunächst soll nur veranschaulicht werden, wie der maximale S-Schnitt auf bestimmte Knoten des EFSNet-Prozesses verweist. Während die Features des Wurzelknotens (Typ PEff) auf das „aktuelle“ Vorkommen einer Transition im Prozeß zeigen, verweisen die Features unter pre auf die entsprechenden „aktuellen“ Marken. Die daraus resultierende Pfadgleichung $s\,\mathsf{m}\,s \doteq \mathsf{pre}\,s$ ist in der Graphnotation in Abbildung 4.26 gut zu erkennen.

Bevor nun ein EFSNet-Prozeß konstruiert wird, werden aus den abgeleiteten Transitionsregeln aus der Schaltfolgensemantik spezielle Prozeß-Transitionsregeln konstruiert, die mit Hilfe des maximalen S-Schnitts einen Prozeß um ein Vorkommen der entsprechenden Transition erweitern können.

Definition 4.30 [Prozeß-Transitionsregelfunktion]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet und $\mathrm{PTS}_{EFSN} = \langle \mathrm{Type_P}, \sqsubseteq_\mathrm{P}, \mathrm{Feat_P}, \mathrm{approp_P} \rangle$ das sich nach Definition 4.23 aus $EFSN$ ergebende Prozeß-Typsystem. Die *Prozeß-Transitionsregelfunktion* $\mathrm{r_p} : \mathrm{T} \to \mathrm{WFS}_{\mathrm{PTS}_{EFSN}}$ werde folgendermaßen konstruiert. Für alle $t \in \mathrm{T}$ sei $s_t \in {\bullet t}$ und $[t]$ eine Feature Structure aus genau einem Wurzelknoten vom Typ $t \in \mathrm{Type}$. Dann sei $\mathrm{r_p}(t)$ definiert als:

$$\mathrm{r_p}(t) := \bigsqcup \left\{ \mathrm{r_a}(t), \right. \qquad \text{(a)}$$

$$
\begin{array}{ll}
[\mathsf{PEff}], & \text{(b)}\\
\bigsqcup_{\mathrm{U}}\{s{::}[\mathsf{Proc}] \mid s \in {\bullet t} \cup t{\bullet}\}, & \text{(c)}\\
s_t\, s_t{::}[t], & \text{(d)}\\
\bigsqcup_{\mathrm{U}}\{s\, s \doteq s'\, s' \mid s, s' \in {\bullet t} \cup t{\bullet}\}, & \text{(e)}\\
\left.\mathsf{postc} \doteq s_t\, s_t\, \mathsf{m}\right\} & \text{(f)}
\end{array}
$$

$\Diamond$

Es sollen wiederum die einzelnen Komponenten des Ausdrucks inhaltlich motiviert werden.

(a) Die Prozeß-Transitionsregel stellt eine Spezialisierung der abgeleiteten Transitionsregel dar. Deshalb wird $\mathrm{r_a}(t)$ als Ausgangspunkt herangezogen und durch Unifikation spezialisiert.

(b) Der Typ des Wurzelknotens wird zu PEff spezialisiert, um die zusätzlichen Features zu erlauben.

(c) Für jede Stelle im Wirkungsbereich der Transition werden ein Knoten vom Typ Proc und eine mit dem Stellennamen beschriftete Kante vom Wurzelknoten zu diesem Knoten angelegt.

(d) Da nach Definition 4.8 eine Transition keinen leeren Wirkungsbereich haben kann, ist s_t definiert und wird benutzt, um einen Knoten vom Typ t der gerade behandelten Transition unter dem Pfad $s_t\, s_t$ zu konstruieren.

(e) Damit das Transitionsvorkommen die Marken von allen Stellen ihres Wirkungsbereichs konsumiert, wird durch die in diesem Schritt konstruierten Pfadgleichungen sichergestellt, daß alle Pfade $s\, s$ in dem im vorherigen Schritt erzeugten Transitionsknoten zusammenlaufen.

(f) Schließlich wird die Kopie der Markierung im Nachbereich vom Transitionsknoten koreferenziert, so daß diese in den Prozeß eingefügt werden kann.

Auch diese Konstruktion wird anhand des bekannten Beispiel-Netzes aus Abbildung 4.10 gezeigt.

Abbildung 4.27 zeigt die Prozeß-Transitionsregel von t1 als AVM und als Graph. Im Graphen sind die Teile, die aus der abgeleiteten Transitionsregel aus Abbildung 4.10 übernommen wurden, grau dargestellt. Dadurch kann man gut erkennen, daß die Prozeß-Transitionsregel um einen Teilgraphen erweitert wird, der für jede Stelle im Wirkungsbereich der Transition diese als Konsumenten in den Prozeß einträgt. Die durch das Transitionsvorkommen abgelegten Marken (Feature m) werden

über das Feature postc vom Wurzelknoten aus zugänglich gemacht, um bei der Konstruktion des Prozesses je nach Semantik ein einfaches Kopieren oder Referenzieren der Folgemarkierung zu erlauben (siehe unten).

Für die übrigen Transitionen und die leicht modifizierte Variante des Netzes aus Abbildung 4.11 kann die Konstruktion analog durchgeführt werden und wird deshalb hier nicht dargestellt.

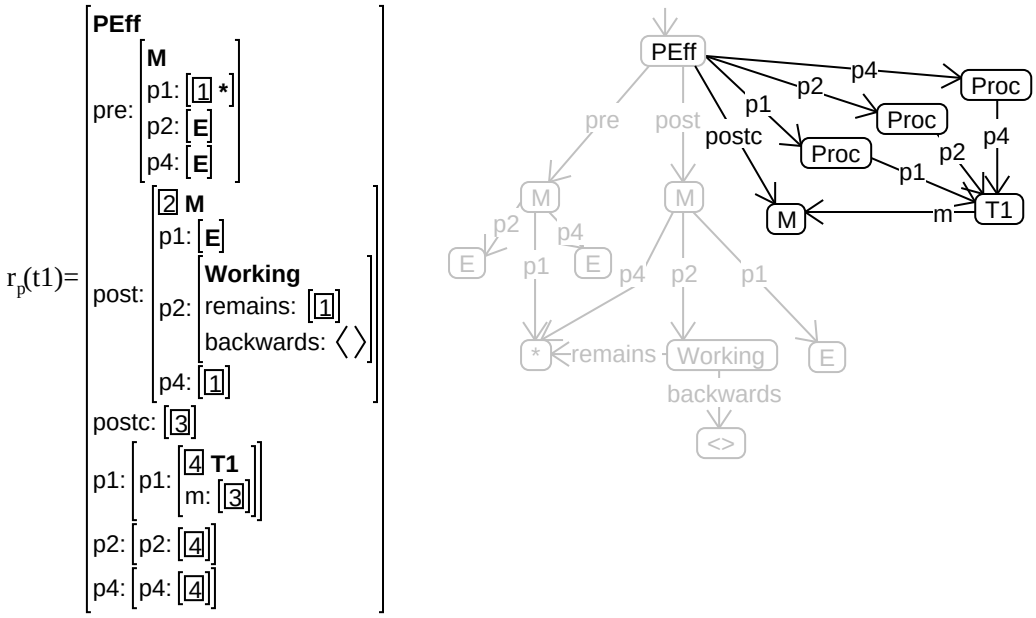

Abbildung 4.27: Die Prozeß-Transitionsregel $r_p(t1)$ der Transition t1 aus Abbildung 4.10 als AVM und als Graph. Der grau dargestellte Teil entspricht der abgeleiteten Transitionsregel aus Abbildung 4.14.

Mit Hilfe der so konstruierten Prozeß-Transitionsregeln kann nun ein EFSNet-Prozeß effektiv konstruiert werden.

Definition 4.31 [Konstruktion eines EFSNet-Prozesses]
Sei $EFSN = \langle \mathrm{S}, \mathrm{T}, \mathrm{TS}, \mathrm{l}, \mathrm{r}, \mathrm{m}_0 \rangle$ ein elementares FSNet. Ein EFSNet-Prozeß $P \in \mathrm{PTS}_{EFSN}$ von $EFSN$ wird nach folgender Konstruktionsvorschrift erstellt.

1. Konstruiere P_0 aus der Anfangsmarkierung von $EFSN$ als

$$P_0 := \left[\begin{array}{l} \mathsf{Proc} \\ \mathsf{m}\colon [\mathsf{M}] \end{array} \right] \sqcup \mathsf{m}{::}\mathrm{m}_0.$$

2. Konstruiere aus dem aktuellen Prozeß P_i den aktuellen maximalen S-Schnitt $\mathrm{scut}(P_i)$.

3. Wähle eine Transition $t \in \mathrm{T}$, für die folgende Feature Structure definiert ist:

$$S_i := \mathrm{scut}(P_i) \sqcup \mathrm{r_p}(t)$$

   Existiert kein solches t, ist der Prozeß nicht mehr erweiterbar und somit das Netz tot. Terminiere mit dem Ergebnis P_i.

4. Gab es ein solches t, erzeuge den Folgeprozeß:

   (a) Bei Referenzsemantik:

$$P_{i+1} := (S_i \sqcup (\mathsf{post} \doteq \mathsf{postc}))@\mathsf{proc}$$

   (b) Bei Wertsemantik:

$$P_{i+1} := (S_i \sqcup \mathsf{postc}::\mathrm{valmark}(S_i@\mathsf{post}))@\mathsf{proc}$$

5. Terminiere mit dem Ergebnis P_i
   oder setze $i := i + 1$ und gehe zu Schritt 2. ◊

Diese Konstruktion soll für das bekannte Beispiel nach Referenzsemantik durchgeführt werden. Für Wertsemantik wäre wie in Schritt 4. (b) angegeben die Folgemarkierung zu kopieren statt zu referenzieren.

Aus Platzgünden finden sich die Darstellungen der einzelnen Schritte der Konstruktion in Anhang C. Abbildung C.1 zeigt den in Schritt 1. konstruierten Anfangsprozeß P_0.

Da in der Referenzsemantik die Pfadwerte unter post und postc identifiziert werden, werden im folgenden vereinfachte Varianten der Prozeß-Transitionsregeln benutzt, welche direkt die vom Vorkommen der Transition erzeugte Folgemarkierung unter post eintragen.

Abbildung C.2 zeigt einen Ausdruck aus dem in Schritt 2. konstruierten maximalen S-Schnitt $\mathrm{scut}(P_0)$ unifiziert mit der Prozeß-Transitionsregel von t1, was in diesem Fall die einzige Möglichkeit darstellt. Der Teilausdurck unter dem Pfad proc ergibt einen Folgeprozeß P_1, zu sehen rechts vom Gleichheitszeichen. Die Knotenreferenz-Bezeichner der Prozeß-Transitionsregeln sind in dieser und allen folgenden Abbildungen so angepaßt, daß die Unifikation einfacher nachzuvollziehen ist und sie von einer Abbildung zur nächsten die sich entsprechenden Feature-Structure-Knoten bezeichnen.

Aus dem Prozeß P_1 wird der maximale S-Schnitt $\mathrm{scut}(P_1)$ erzeugt. Auf diesen ist die Prozeß-Transitionsregel von t2 oder von t4 anwendbar. Abbildung C.3 zeigt die Anwendung von t2. Es ergibt sich der Prozeß P_2.

In dieser Art wird der Prozeß schrittweise mit den Prozeß-Transitionsregeln von t4 (Abbildung C.4), t2 (Abbildung C.5) und schließlich t3 (Abbildung C.6) erweitert. Dieser letzte Prozeß ist nicht mehr erweiterbar. Wie im Erreichbarkeitsgraphen

existieren zwei weitere nicht dargestellte Teilprozesse, die den zwei weiteren Schaltzeitpunkten von t4 entsprechen.

Mit dem Schalten von t4 ergibt sich ein besonders interessanter Fall. t4 erzeugt nicht neue Marken, die auf bereits existierende verweisen, sondern verändert eine existierende Marke, indem Typinformation (A) hinzugefügt wird. Durch die Referenzsemantik ist diese Veränderung für alle anderen Transitionen sichtbar. Anhand des letzten Prozesses P_5 ist der Zeitpunkt nicht mehr feststellbar, an dem die Information hinzugefügt wurde, daß das Listenelement vom Typ A ist. Dies entspricht zwar nicht der üblichen Bedeutung von S-Schnitten in Prozessen, mögliche Markierungen des Netzes darzustellen (siehe Abschnitt 3.2.2, kann aber folgendermaßen interpretiert werden. Feature Structures stellen die Information dar, die über ein Objekt vorliegt. Der Zeitpunkt, an dem Information hinzugefügt wird, ist in einem Prozeß irrelevant, wenn diese Information so vorliegen muß, damit der Prozeß in dieser Weise stattfinden kann. Es ist nicht zu unterscheiden, ob ein Transitionsvorkommen überprüft, ob bestimmte Information vorhanden ist (Knoten fst war bereits vorher vom Typ A) oder diese hinzufügt. In gewissem Sinne war der Knoten fst „schon immer“ vom Typ A, wenn der Prozeß so verläuft wie gezeigt. Wichtig ist allerdings, daß in einem Prozeß keine unnötige, das heißt von keiner Anfangsmarkierung oder schaltenden Transition spezifizierte Information hinzugefügt wird. Deshalb fordert die algebraische Definition des Prozesses, daß dieser die allgemeinste Feature Structure ist, die alle sonstigen Kriterien erfüllt. Andernfalls bestünde die bereits bei der Diskussion der Schaltregel erwähnte Gefahr, daß Information „hinzuerfunden“ wird.

In den folgenden Abbildungen 4.28 und 4.29 sind die Prozesse (hier ohne die S-Schnitte) nochmals auf einen Blick in Graphnotation angegeben. Dabei sind zur besseren Unterscheidung Kanten, welche die Kausalitätsstruktur des Prozesses betreffen, durchgehend dargestellt, während Kanten, welche die Markierung betreffen, gestrichelt gezeichnet sind. Aus Platzgründen sind die Features remains und backwards hier zu rem und back abgekürzt. Um den Bezug zu Abbildung C.6 herzustellen, sind – für die Graphnotation unüblich – die Knotenreferenzen aus der AVM-Notation den Knoten vorangestellt. Zur Unterscheidung zwischen Namen und Typen wird die UML-Notation herangezogen und den Typen ein Doppelpunkt vorangestellt. Die Referenzsemantik des Prozesses ist anhand der gestrichelten Kanten, die Marken-Knoten von verschiedenen Marken aus koreferenzieren, sehr gut zu erkennen.

Durch die gleichzeitige Darstellung von Kausalitäten und Marken-Werten wird die Darstellung allerdings recht unübersichtlich. Abbildung 4.30 zeigt deshalb eine eingeschränkte Sicht auf den letzten Prozeß, in der die Markierungs-Substrukturen (m-Features) weggelassen wurden. Hier sind die kausalen Abhängigkeiten wie bei klassischen Prozessen von S/T-Netzen klar zu erkennen.

Ein ähnliches Bild ergibt sich für Wertsemantik-Prozesse. Hier entfallen die ausdrucksstarken, aber schwer lesbaren Referenzen zwischen verschiedenen Marken. Ab-

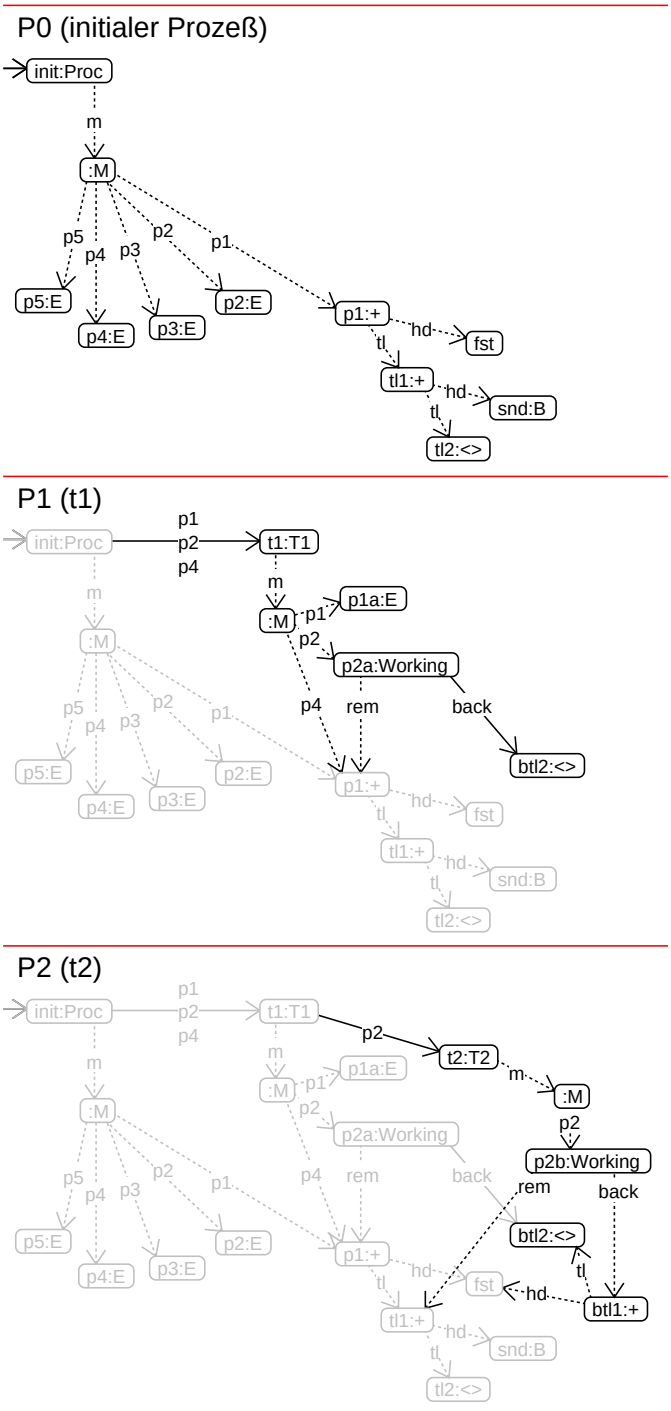

Abbildung 4.28: Die einzelnen Schritte der Erstellung eines Referenzsemantik-Prozesses des Netzes aus Abbildung 4.10.

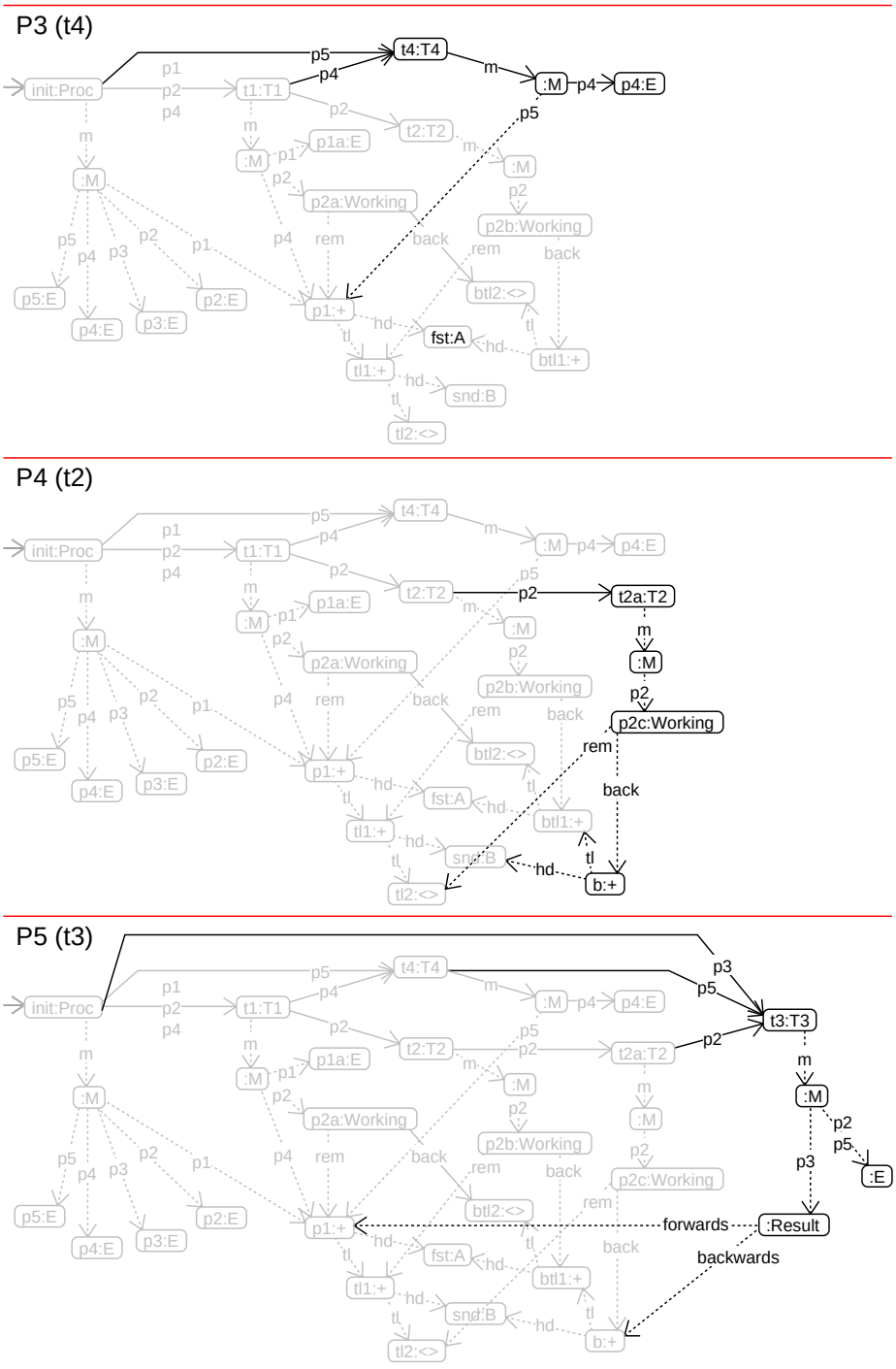

Abbildung 4.29: Die einzelnen Schritte der Erstellung eines Referenzsemantik-Prozesses des Netzes aus Abbildung 4.10 (Fortsetzung).

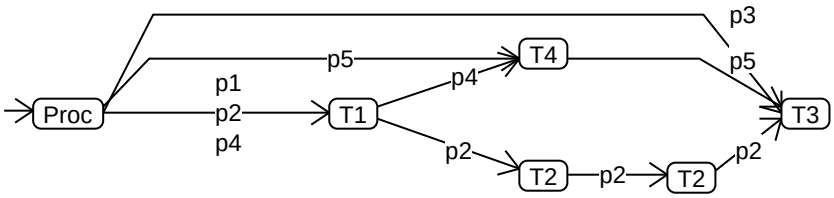

Abbildung 4.30: Eine auf kausale Abhängigkeiten reduzierte Sicht des Prozesses P_5 aus den Abbildungen C.6 und 4.29.

bildung 4.31 zeigt einen Wertsemantik-Prozeß des Netzes aus Abbildung 4.11, der dem Referenzsemantik-Prozeß aus Abbildung C.6 stark ähnelt. In der Darstellung wird auf die Mischnotation für Feature Structures aus Abschnitt 2.3.1 zurückgegriffen. Es ist deutlich zu erkennen, daß zwischen den Marken keine Koreferenzen existieren, wohl aber innerhalb einzelner Marken.

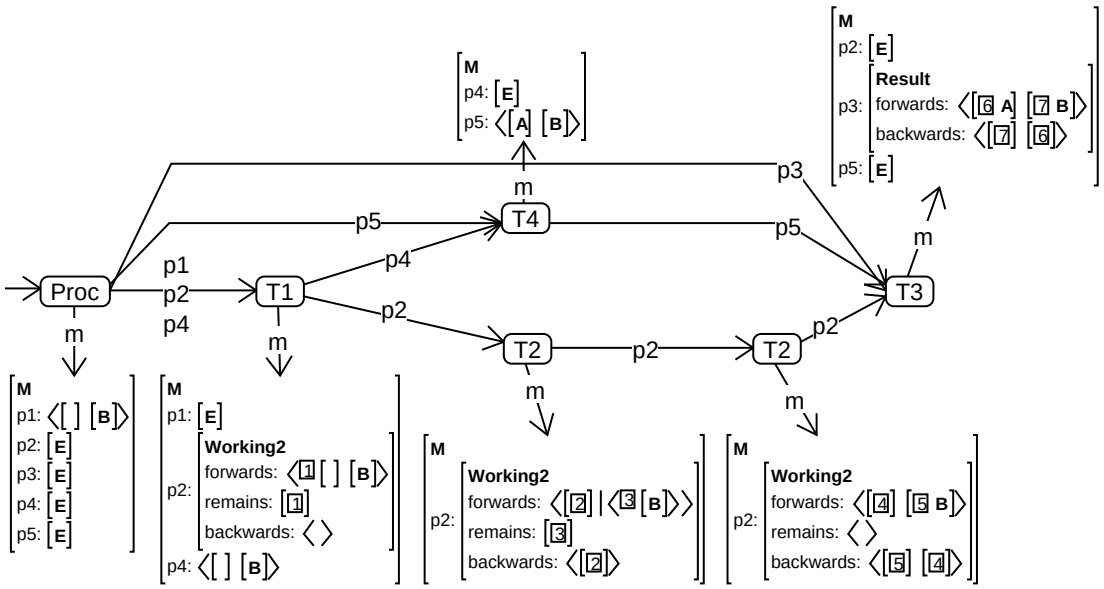

Abbildung 4.31: Ein Wertsemantik-Prozeß des Netzes aus Abbildung 4.11.

Durch die konstruktive Definition von Prozessen erhalten elementare FSNets eine echte Nebenläufigkeitssemantik. Die in diesem Abschnitt vorgestellten Konstruktionsalgorithmen wurden implementiert und in das Werkzeug FSRenew integriert, so daß mit der Prozeßsemantik anhand konkreter, auch größerer Beispiele experimentiert werden kann. Einzelheiten zur Implementierung enthält Abschnitt 4.5.

Mit der Definition einer Prozeßsemantik für EFSNets sind die Voraussetzungen geschaffen, um Netze in Netzen und FSNets zu verbinden. Der folgende Abschnitt widmet sich diesem Thema.

4.3 Netze in Netzen als FSNets

Im vorherigen Abschnitt wurden für elementare FSNets Prozesse als Feature Structures dargestellt. Die in Abschnitt 3.3.1 vorgestellten elementaren Objektsysteme von Valk verwenden Prozeßmarkierungen, um das Konzept von Netzen in Netzen umzusetzen. Wenn nun FSNets beliebige Feature Structures als Marken erlauben und Prozesse als Feature Structures dargestellt werden können, liegt es nahe, Valks prozeßmarkierte Systemnetze als FSNets zu formulieren.

Dafür gibt es zwei grundsätzlich verschiedene Möglichkeiten: Entweder, System- und Objektnetz(e) werden gemeinsam von einem FSNet simuliert, oder es wird eine spezielle Form des EFSNet-Prozesses definiert, die wie in [Valk 1996] einen gemeinsamen Prozeß von System- und Objektnetz definiert und damit eine Prozeßsemantik für Netze in Netzen darstellt. Beide Ansätze sollen hier weiter verfolgt werden. Eine formale Definition würde aber über den Rahmen dieser Arbeit hinausgehen, weshalb die Ansätze stattdessen detailliert beschrieben und an Beispielen motiviert werden.

4.3.1 System- und Objektnetz als ein FSNet

Im ersten Ansatz werden System- und Objektnetz(e) gemeinsam von einem FSNet simuliert. Hier ist als Struktur des Systemnetzes ein beliebiges FSNet erlaubt. Das Objektnetz wird auf ein elementares FSNet beschränkt, da nur für diese Netzklasse eine Prozeßsemantik definiert wurde. Dies ist eine Erweiterung gegenüber den EOS, da dort für Objektnetze nur B/E-Netze erlaubt sind, EFSNets hingegen eine Färbung der Marken durch beliebig komplexe Feature Structures erlauben. Das FSNet spiegelt die Struktur des Systemnetzes wieder, während die Struktur des Objektnetzes durch dessen Prozeß-Transitionsregeln (siehe Abschnitt 4.2.8) als Marken im Systemnetz repräsentiert wird.

Alle weiteren Feature-Structure-Marken im Systemnetz stellen Prozesse des Objektnetzes dar, was der p-Markierung bei Valk (siehe Abschnitt 3.3.1) entspricht. Das FSNet übernimmt damit nicht nur die Simulation des Systemnetzes, sondern auch die des Objektnetzes oder der Objektnetze. Durch die Synchronisationsrelation ρ ist bekannt, welche Transitionen des Systemnetzes eine Transitionsregel auf einen Prozeß des Objektnetzes anwenden *müssen*. Da eine Feature Structure in der normalen Schaltregel für FSNets kein aktives Objekt darstellt, müssen die objektautonomen Schaltvorgänge durch zusätzliche Transitionen für jede Stelle simuliert werden. Diese wenden genau eine der Transitionsregeln auf Objektnetz-Prozesse an, die keine Synchronisation erfordern.

Ein Basis-FSNet reicht zur Anwendung einer Prozeß-Transitionsregel auf einen Prozeß *nicht* aus. Zwar erlauben Basis-FSNets die in der Konstruktion in Definition 4.31 benötigten Operationen Unifikation und Pfadwert-Bildung, die Berechnung des maximalen S-Schnitts (scut) ist aber nicht möglich. Als Alternative könnte der maximale S-Schnitt genau wie der Prozeß von einem Schritt zum nächsten inkrementell berechnet werden, was durch ein Basis-FSNet geleistet werden könnte. Dies würde von einer p-Markierung zu einer scut-Markierung führen. In Valks Ansatz werden Prozesse aber nicht nur fortgeschrieben, sondern auch unifiziert. Die Darstellung von EFSNet-Prozessen als Feature Structures wurde so gewählt, daß die Feature-Structure-Unifikation der Prozeß-Unifikation von Valk entspricht. Anhand eines einfachen Beispiels kann sich der Leser klar machen, daß dagegen die Unifikation maximaler S-Schnitte nicht sinnvoll ist. Auch wenn diese unifizierbare Prozesse enthalten, führt die Unifikation der maximalen S-Schnitte nicht zum gewünschten Ergebnis. Da für jede Stelle in den maximalen S-Schnitten auf unterschiedliche Knoten des Prozesses verwiesen werden kann, würden diese bei der Unifikation identifiziert. Demnach wird das korrekte Ergebnis nur erzielt, indem Prozesse unifiziert werden und aus dem Ergebnis der Unifikation der maximale S-Schnitt rekonstruiert wird. An dieser Stelle ist also auf jeden Fall die Anwendung der Funktion scut nötig. Aus diesem Grunde wurde auf eine inkrementelle Konstruktion der maximalen S-Schnitte verzichtet.

Valk stellt in [Valk 1998] das gegenüber dem allgemeinen EOS leicht vereinfachte Modell des *einfachen EOS* (*simple EOS*) auf und gibt dort nur für dieses eine formale Schaltregel an. Für die Simulation von einfachen EOS reichen Basis-FSNets aus, da hier keine Prozesse unifiziert werden. Dieser Fall wird hier aber nicht weiter betrachtet.

Die folgenden detaillierteren Ausführungen können anhand des konkreten Beispiels in Abbildung 4.32 nachvollzogen werden. Als unäres EOS wird das bekannte Beispiel aus Abbildung 3.19 herangezogen.

Es gibt für jede Interaktionsanschrift eine Stelle, in welche die Transitionsregeln der entsprechenden Transitionen einsortiert werden. Transitionsregeln von Transitionen ohne Interaktion werden in der Stelle `no_i` abgelegt. In diesem einfachen Fall hätten die Transitionsregeln auch direkt an die Transitionen des Systemnetzes geschrieben werden können, da es zu jeder Interaktion genau eine Transition gibt (und mit `e5` auch genau eine Transition ohne Interaktion). In komplexeren Beispielen werden in den Stellen aber mehrere Marken liegen, zwischen denen die Transition des Systemnetzes auswählen kann. Auch hier könnte man die Interaktionsstellen durch entsprechend viele Versionen der Transition ersetzen. Dies wurde aber nicht getan, da nichts dagegen spricht, die Vorteile der Färbung des Netzes auszunutzen und weiterhin so eine stärkere Trennung von System- und Objektnetz vorliegt. Soll ein anderes Objektnetz verwendet werden, so ist in der gewählten Modellierung nur die Markierung, nicht die Struktur des Systemnetzes zu ändern. Ein weiterer Vorteil der gewählten Modellierung ist, daß diese die Möglichkeit eröffnet, das Objektnetz

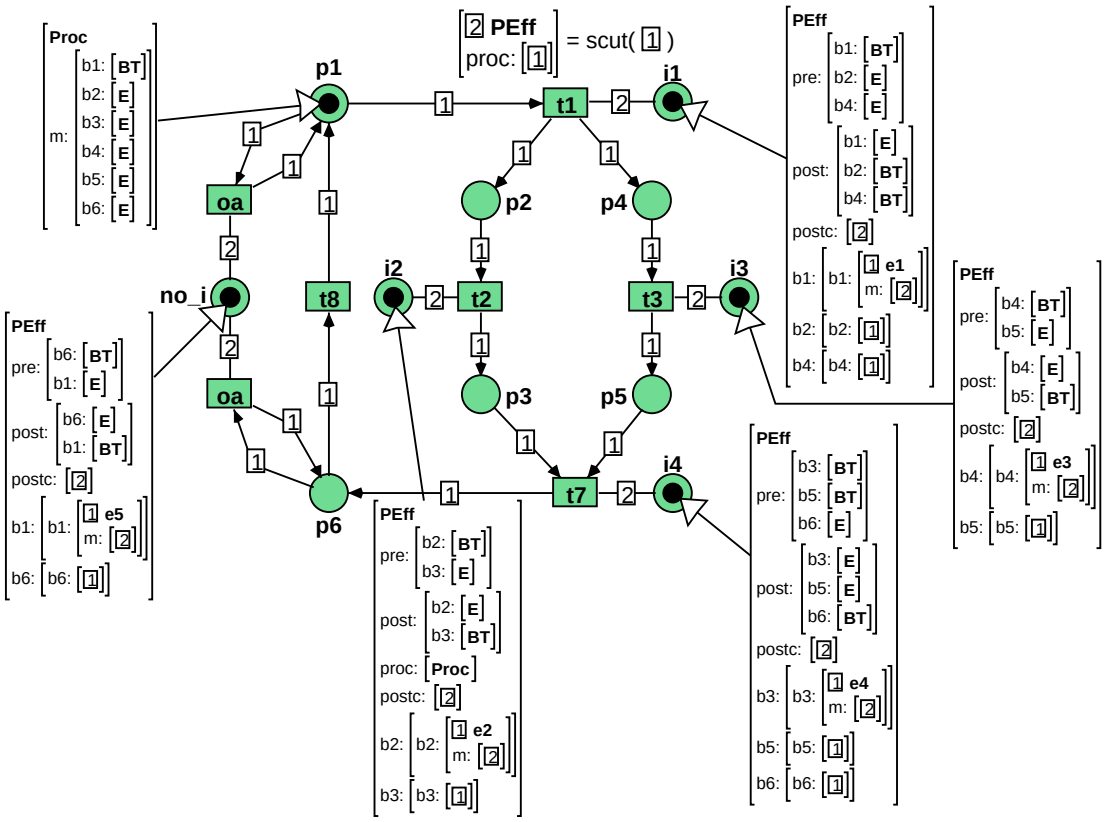

Abbildung 4.32: Das EOS aus Abbildung 3.19 als FSNet.

zur Laufzeit zu modifizieren. Solche dynamischen Änderungen von Netzstrukturen wurden bereits in [Farwer 2000b] vorgeschlagen (siehe Abschnitt 3.3.2).

Die Anfangsmarkierung des FSNets ergibt sich folgendermaßen. Jede Stelle, die in der Anfangsmarkierung des EOS-Systemnetzes markiert ist, wird mit dem initialen Prozeß des Objektnetzes markiert, der sich aus der Anfangsmarkierung des Objektnetzes und der Prozeßkonstruktion für EFSNets aus Abschnitt 4.2.8 ergibt. Da es sich bei EOS-Objektnetzen um elementare Netzsysteme handelt, wird für das EFSNet außer dem Typ E für unmarkierte Stellen nur ein weiterer Typ für das *black token* benötigt, der hier BT genannt wurde.

Transition t1 trägt eine Anschrift, welche die Funktion scut mit der durch 1 gebundenen Eingangsmarke als Parameter aufruft[6] und das Ergebnis mit der links vom Gleichheitszeichen angegebenen Feature Structure unifiziert. Die gleiche Anschrift muß an *allen* Transitionen des Systemnetzes angegeben werden, wurde aber

[6]Der Aufruf von Funktionen in Transitionsanschriften ist eine Erweiterung von FSNets, die über die Definition von Basis-FSNets hinausgeht und in Abschnitt 4.5 genauer betrachtet wird.

aus Gründen der Übersichtlichkeit bei den anderen Transitionen in der Abbildung weggelassen.

Die oben erwähnten Hilfstransitionen, die für objektautonomes Schalten des Objektnetzes sorgen, sind in der Abbildung mit `oa` beschriftet. Im allgemeinen Fall muß es eine solche Transition für jede Stelle des Systemnetzes geben. Im Beispiel ist nur in den Stellen `p1` und `p6` autonomes Schalten möglich, weswegen die für die übrigen Stellen zuständigen `oa`-Transitionen als Optimierung weggelassen wurden, was gleichzeitig das Netz besser lesbar macht. Im Beispiel kann das autonome Schalten von `e5` vor oder nach dem Transport durch `t8` stattfinden.

Die Konstruktion eines solchen FSNets aus einem gegebenen EOS kann offensichtlich automatisiert werden. Wie die Prozeß-Transitionsregeln des Objektnetzes erstellt werden, geht aus Definition 4.30 hervor. Die regelbasierte Konstruktion des FSNets aus dem Systemnetz und diesen Transitionsregeln für die Anfangsmarkierung geht (zumindest informal) aus dem oben vorgestellten Beispiel hervor.

Durch die Erweiterungen von höheren FSNets (Abschnitt 4.4) wird gegenüber Basis-FSNets unter anderem der Aufruf beliebiger Java-Methoden erlaubt. Die Funktion scut wurde in Java implementiert, so daß Netze in Netzen nach dem hier diskutierten Ansatz mit dem im Rahmen dieser Arbeit erstellten Werkzeug simuliert werden können.

Im folgenden Abschnitt wird der zweite Ansatz zur Darstellung von Netzen in Netzen mit FSNets vorgestellt. Im übernächsten Abschnitt werden dann beide Ansätze diskutiert und verglichen.

4.3.2 Feature-Structure-Prozesse von Netzen in Netzen

Der zweite Ansatz stellt den Ablauf von Netzen in Netzen als einen erweiterten EFSNet-Prozeß dar. Hier wird die Struktur für Objekt- *und* Systemnetz auf elementare FSNets beschränkt, da für beide Prozesse dargestellt werden sollen. Dies entspricht Valks Ansatz der unären EOS, stellt aber insofern eine Erweiterung dar, als die Objektnetze beliebige Feature Structures als Marken enthalten können. Es ergeben sich demnach *gefärbte Prozesse von Netzen in Netzen*.

Das Systemnetz wird im Gegensatz zu dem im vorherigen Abschnitt vorgestellten Ansatz als *elementares* FSNet dargestellt. Da selbst Basis-FSNets nicht ausreichen, um das Schaltverhalten des Objektnetzes zu simulieren (s.o.), muß bei dieser Darstellung eine neue Schaltregel definiert werden, welche die Anwendung der Funktion scut auf einzelne Marken einschließt. Im folgenden wird auf diese Schaltregel nicht weiter eingegangen, sondern gleich eine Prozeßsemantik vorgestellt. Die Schritte zur Konstruktion des Prozesses, die unten gegeben werden, stellen im Prinzip eine Schaltregel dar.

Die Prozeßsemantik eines elementaren FSNet basiert auf einem Typsystem und einer Menge von Prozeß-Transitionsregeln, die aus der Struktur des Netzes abgeleitet werden. Dementsprechend gibt es in der Prozeßsemantik für Netze in Netzen

ein Typsystem, das sich aus dem Prozeß-Typsystem für System- und Objektnetz zusammensetzt, sowie für jedes der Netze ein Menge von Prozeß-Transitionsregeln. Analog zu Valks elementaren Objektsystemen wird der Verbund aus System- und Objektnetz zusammen mit der Interaktionsrelation ein EFSNet-Objektsystem genannt.

Definition 4.32 [EFSNet-Objektsystem]
Ein *EFSNet-Objektsystem* ist ein Tupel $\langle SN, ON, \rho \rangle$ aus folgenden Komponenten:

- einem EFSNet $ON = \langle \mathrm{S}_{ON}, \mathrm{T}_{ON}, \mathrm{TS}_{ON}, \mathrm{l}_{ON}, \mathrm{r}_{ON}, \mathrm{m}_{0,ON} \rangle$ nach Definition 4.8,
- einem EFSNet $SN = \langle \mathrm{S}_{SN}, \mathrm{T}_{SN}, \mathrm{TS}_{SN}, \mathrm{l}_{SN}, \mathrm{r}_{SN}, \mathrm{m}_{0,SN} \rangle$, dessen Typsystem das Prozeß-Typsystem von TS_{ON} nach Definition 4.23 ist ($\mathrm{TS}_{SN} = \mathrm{PTS}_{\mathrm{TS}_{ON}}$), dessen Transitionsregeln alle der allgemeinsten Feature Structure entsprechen ($\forall t \in \mathrm{T}_{SN} : \mathrm{r}(t) \sim [\top]$) und das keine Transitionen ohne Vorbedingungen enthält ($\forall t \in \mathrm{T}_{SN} : \exists s \in \mathrm{S}_{SN} : \mathrm{l}_{SN}(s,t)$ ist definiert) und
- einer *Interaktionsrelation* $\rho \subseteq \mathrm{T}_{SN} \times \mathrm{T}_{ON}$ zwischen SN und ON. ◊

Das Typsystem des Systemnetzes entspricht dem Prozeß-Typsystem des Objektnetzes, so daß Prozesse des Objektnetzes erlaubte Marken im Systemnetz darstellen. Das Systemnetz selbst ist ein triviales EFSNet insofern, als es nur „leere“ Transitionsregeln enthält. Der Sinn dieser Einschränkung ist, daß ein Systemnetz bei EOS die Objektnetzprozesse nicht manipuliert, sondern nur unifiziert. Dieser Effekt wird genau durch die leere Transitionsregel erreicht. Daß alle definierten Kantenanschriften aus dem leeren Pfad bestehen müssen, braucht nicht extra gefordert zu werden, da ein EFSNet nur dann wohlstrukturiert definiert ist, wenn die Pfade in der Transitionsregel vorkommen. Außerdem muß jede Transition im Systemnetz mindestens eine Vorbedingung besitzen, da sonst nicht klar wäre, was für eine Prozeßmarke im Nachbereich abgelegt werden soll. Man könnte auch definieren, daß in einem solchen Fall wie bei der Anfangsmarkierung der initiale Prozeß des Objektnetzes erzeugt wird.

Das EFSNet-Objektnetz kann hingegen ein beliebiges EFSNet sein. Hierin liegt die Erweiterung gegenüber den EOS.

Auch für das Systemnetz soll ein Prozeß-Typsystem erstellt werden. Dabei müssen die mit konstanten Namen versehenen Typen umbenannt werden, um eine Verwechslung mit den Typen des Objektnetz-Prozeß-Typsystems zu vermeiden. Wir notieren diese Unterscheidung durch einen Index SN bzw. ON am Typnamen. Ansonsten gleicht die Konstruktion der in Definition 4.23 und wird hier nicht angepaßt wiedergegeben. Abbildung 4.33 zeigt als Beispiel für ein Prozeß-Typsystem eines EFSNet-Objektsystems die Konstruktion für das Beispiel aus Abbildung 3.19. Die Indizes sind aus darstellungstechnischen Gründen als Postfixe _SN bzw. _ON notiert. Wie man sieht, wird das Prozeß-Typsystem des Objektnetzes als Markentyp des Systemnetzes verwendet. Unter dem Typ $\top_{ON}$ könnten weitere Subtypen für

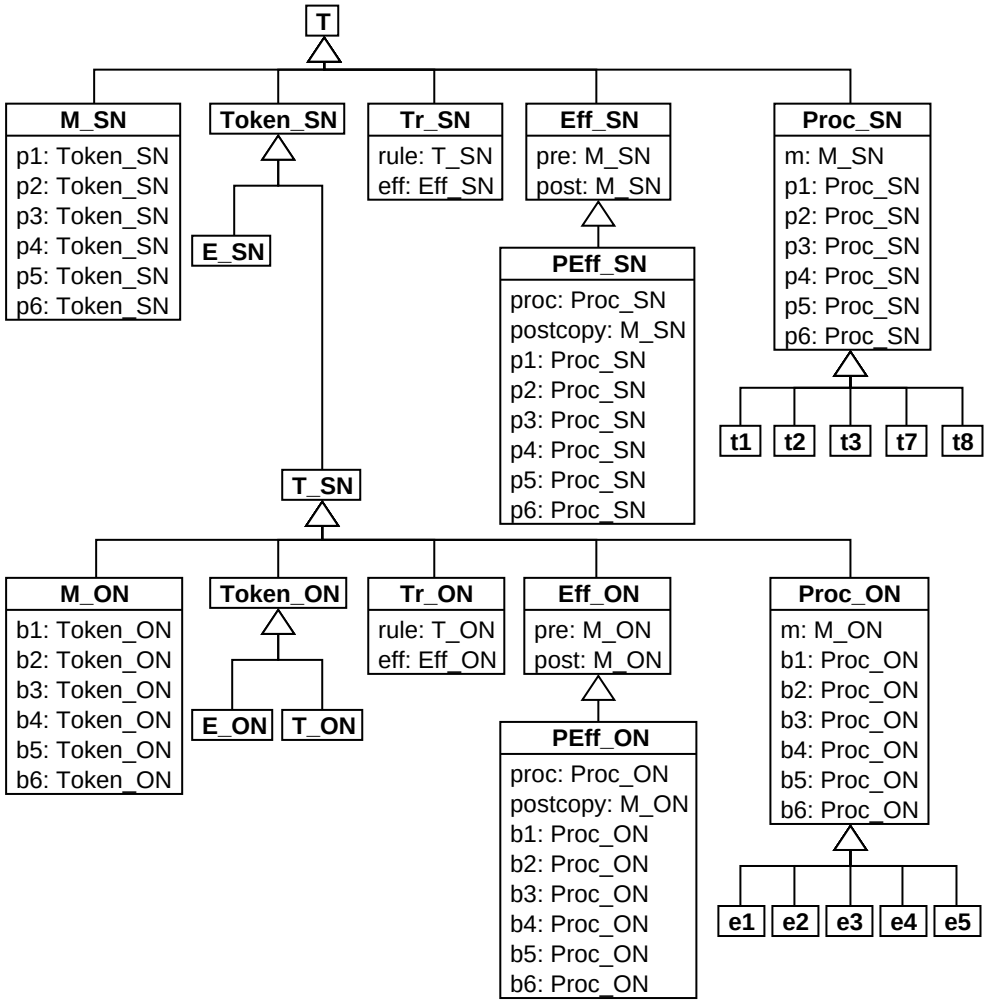

Abbildung 4.33: Das Prozeß-Typsystem des EOS aus Abbildung 3.19.

die Marken des Objektnetzes definiert werden. Da es sich bei dem Beispiel von Valk aber um ungefärbte Netze handelt, gibt es hier nur den Typ für die anonyme Marke, der dem Typ BT aus dem vorherigen Abschnitt entspricht.

Die Konstruktion eines EFSNet-Prozesses zu einem EFSNet-Objektsystem greift auf den Algorithmus aus Definition 4.31 in Abschnitt 4.2.8 zurück, in dem EFSNet-Prozesse eines elementaren FSNets erstellt werden. Zur Nachbildung der Standardsemantik von EOS wird die Wertsemantik-Variante (b) genutzt. Der Prozeß eines wie in Definition 4.32 gegebenen EFSNet-Objektsystems wird wie folgt konstruiert:

- Alle Stellen, die im Systemnetz SN eine Anfangsmarkierung tragen, werden mit dem initialen Prozeß markiert, der sich aus der Anfangsmarkierung des

Objektnetzes ON ergibt. Alle anderen Stellen von SN werden mit E_{SN} markiert. Aus dieser Anfangsmarkierung von SN ergibt sich der initiale Prozeß des EFSNet-Objektsystems.

- Der maximale S-Schnitt $S_{SN,i}$ zum Prozeß P_i wird durch die Funktion scut konstruiert: $S_{SN,i} := \mathrm{scut}(P_i)$. Für die Konstruktion eines Folgeprozesses P_{i+1} gibt es wie bei EOS drei Möglichkeiten (siehe Abschnitt 3.3.1, Abbildung 3.21):

  (a) Das Systemnetz schaltet autonom. Wähle eine Systemnetztransition $t_{SN} \in \mathrm{T}_{SN}$, die nicht in Interaktionsrelation zu einer Objektnetztransition steht ($\neg\exists t_{ON} \in \mathrm{T}_{ON} : (t_{SN}, t_{ON}) \in \rho$). Es gilt analog zu der normalen Prozeßkonstruktion nach Wertsemantik

  $$P_{i+1} := (S_{SN,i} \sqcup \mathsf{postc}{::}\mathrm{valmark}((S_{SN,i} \sqcup \mathrm{r}_{SN,\mathrm{p}}(t_{SN}))@\mathsf{post}))@\mathsf{proc}.$$

  (b) Systemnetz und Objektnetz interagieren. Wähle eine Systemnetztransition $t_{SN} \in \mathrm{T}_{SN}$ und eine Objektnetztransition $t_{ON} \in \mathrm{T}_{ON}$, für die gilt $(t_{SN}, t_{ON}) \in \rho$ und für die der Prozeß P_i folgendermaßen erweitert werden kann. Durch die triviale Struktur des Systemnetzes führen in der Prozeß-Transitionsregel $\mathrm{r}_{SN,\mathrm{p}}(t_{SN})$ alle Pfade $\mathsf{pre}\, s$ mit $s \in {\bullet t_{SN}}$ zum selben Knoten. Sei π einer dieser Pfade. Der Ausdruck $P_{ON,i} := (S_{SN,i} \sqcup \mathrm{r}_{SN,\mathrm{p}}(t_{SN}))@\pi$ führt also zur Unifikation aller Prozeßmarken im Vorbereich von t_{SN}. Aus $P_{ON,i}$ wird wie im folgenden Punkt beschrieben der Folgeprozeß $P_{ON,i+1}$ des Objektnetzes für ein Schalten von t_{ON} konstruiert. Der Folgeprozeß des EFSNet-Objektsystems ergibt sich zu

  $$P_{i+1} := (S_{SN,i} \sqcup \mathsf{postc}{::}\mathrm{valmark}((S_{SN,i} \sqcup \mathrm{r}_{SN,\mathrm{p}}(t_{SN}) \sqcup \pi :: P_{ON,i+1})@\mathsf{post}))@\mathsf{proc}.$$

  (c) Das Objektnetz schaltet autonom. Wähle eine Transition $t_{ON} \in \mathrm{T}_{ON}$, die nicht in Interaktionsrelation zu einer Systemnetztransition steht ($\neg\exists t_{SN} \in \mathrm{T}_{SN} : (t_{SN}, t_{ON}) \in \rho$) und ein Feature $s \in \mathrm{S}_{SN}$, für das gilt $P_{ON,i} := S_{SN,i}@s \not\sim [\mathsf{E}_{SN}]$. Weiterhin muß der maximale S-Schnitt des gewählten Objektnetzprozesses mit der Prozeß-Transitionsregel von t_{ON} unifizierbar sein. Sei $S_{ON,i} := \mathrm{scut}(P_{ON,i}) \sqcup \mathrm{r}_{\mathrm{p},ON}(t_{ON})$. Es gilt analog zu der normalen Konstruktion nach Wertsemantik

  $$P_{ON,i+1} := (S_{ON,i} \sqcup \mathsf{postc}{::}\mathrm{valmark}(S_{ON,i}@\mathsf{post}))@\mathsf{proc}.$$

  Der Prozeß des EFSNet-Objektsystems ergibt sich zu

  $$P_{i+1} := (S_i \sqcup s{::}P_{ON,i+1})@\mathsf{proc}.$$

  Da ein Prozeß immer seinen Vorgängerprozeß spezialisiert ($P_{ON,i} \sqsubseteq P_{ON,i+1}$), kann diese letzte Unifikation nicht scheitern.

Als einen Sonderfall kann man beim systemautonomen Schalten den reinen Transport unterscheiden, in dem genau eine Prozeßmarke von einer Stelle auf eine andere bewegt wird. Hier kann man unterschiedlich definieren, ob eine Kopie erzeugt oder die Prozeßmarke beim Transport erhalten wird. Im letzteren Fall erhält man eine noch höhere Nebenläufigkeit (siehe unten). Man kann diesen Fall in der Konstruktion oben realisieren, indem in Fall (a) für den Sonderfall $|\bullet t_{SN}| = |t_{SN}\bullet| = 1$ Referenzsemantik angenommen wird, also der Teilausdruck mit der Funktion valmark wie in der Konstruktion 4.31 durch die Pfadgleichung post $\doteq$ postc ersetzt wird.

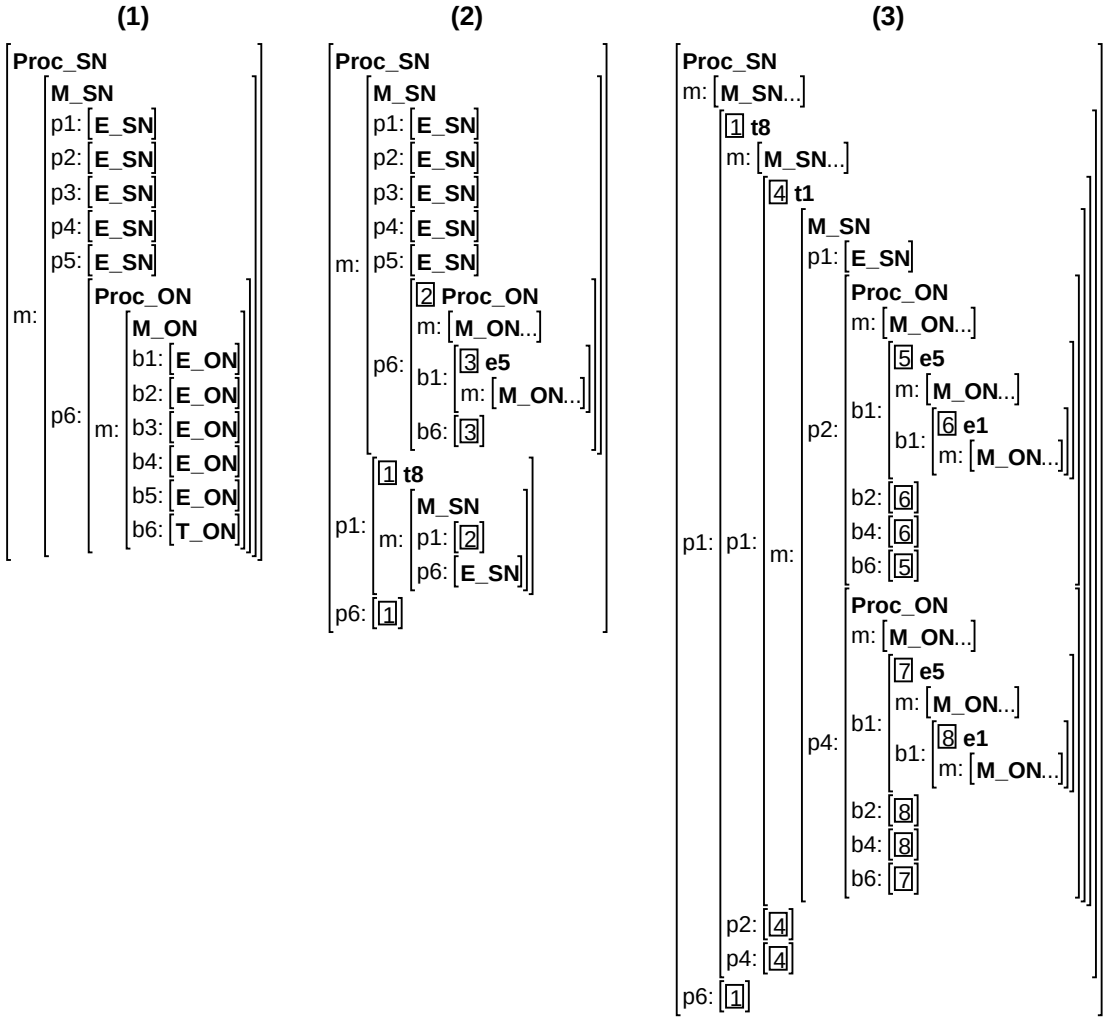

Abbildung 4.34: Feature-Structure-Prozesse des EOS aus Abbildung 3.19 über dem Typsystem aus Abbildung 4.33 als AVMs: (1) initialer Prozeß, (2) t8 und e5 schalten autonom, (3) t1 und e1 schalten synchron.

Im folgenden wird vorgeführt, wie ein EFSNet-Prozeß zu dem EFSNet-Objektsystem zum EOS aus Abbildung 3.19 konstruiert wird. Dabei verwenden wir das Typsystem aus Abbildung 4.33. Wir beginnen mit dem initialen Prozeß, der sich nach der angegebenen Konstruktion als ein initialer Prozeß des Systemnetzes ergibt, der in allen markierten Stellen eine Kopie des initialen Prozesses des Objektnetzes enthält, hier also in p6. Abbildung 4.34 (1) zeigt den initialen Prozeß des EFSNet-Objektsystems.

In den folgenden Abbildungen werden für die Objektnetzprozesse die Markierungen, also die Substrukturen unter den Features m, nur angedeutet oder ganz weggelassen. Da es sich bei dem Objektnetz des Beispiels um ein B/E-Netz handelt, das keine Nebenbedingungen enthält, wird von jeder Transition genau die Menge der Bedingungen in ihrem Wirkungsbereich negiert. Da sich der Wirkungsbereich einer Transition aus ihrer Transitionsregel ablesen läßt, ergibt sich die aktuelle Markierung bei gegebener Anfangsmarkierung eindeutig aus dem Prozeß.

Der initiale Prozeß kann durch ein Transitionsvorkommen von t8 des Systemnetzes oder e5 des Objektnetzes fortgesetzt werden. Je nachdem, welche Semantik für den oben beschriebenen Sonderfall des Transports gewählt wird, ergeben sich abhängig von der Reihenfolge der Erweiterung des Prozesses um $< \mathsf{e5}, \mathsf{t8} >$ oder $< \mathsf{t8}, \mathsf{e5} >$ die gleichen (Referenzsemantik) oder unterschiedliche Prozesse (Wertsemantik). Wir wählen hier Referenzsemantik für Transporte und erhalten so unabhängig von der Reihenfolge für $\{\mathsf{e5}, \mathsf{t8}\}$ den Prozeß in Abbildung 4.34 (2).

Nun kann der Prozeß nur fortgesetzt werden, indem t1 und e1 synchron schalten (Interaktionsanschrift $<$ i1 $>$). Diese Interaktion wird wie oben beschrieben durch Anwendung von Objektnetz- und Systemnetz-Transitionsregel in einem Schritt durchgeführt. Abbildung 4.34 (3) zeigt den resultierenden Prozeß. Die Prozeßmarken wurden erst unifiziert, dann das Ergebnis der Unifikation um das Schalten von e1 ergänzt und schließlich kopiert, so daß unter den Features p2 und p4 von der Markierung von t1 gleiche, aber nicht identische Substrukturen erzeugt werden.

Bereits der letzte Prozeß wird in der AVM-Darstellung trotz Auslassungen unübersichtlich. Wie in Abschnitt 4.2.8 bevorzugen wir deshalb die Darstellung von Feature-Structure-Prozessen in Graphnotation, da diese die Ähnlichkeit zu Auftragssystemen betont.

Abbildung 4.35 zeigt einen erweiterten Prozeß des Beispiels, der genau einen Durchlauf der „Schleife“ in System- und Objektnetz repräsentiert. Zur Verdeutlichung wurden die Substrukturen, die Prozeßmarken repräsentieren, grau hinterlegt. In dieser Darstellung ist leicht zu sehen, daß es sich bis auf die Koreferenz auf die erste Prozeßmarke um eine Wertsemantik handelt. Die Koreferenz kommt durch die oben diskutierte Sonderbehandlung des Transports zustande. Einen weiteren interessanten Fall, der in den ersten drei Prozessen aus Abbildung 4.34 nicht auftritt, stellt das Schalten von t7 dar. Die Unifikation der Prozeßmarken der Stellen p3 und p5 zur resultierenden Prozeßmarke auf p6 ist deutlich zu erkennen. In der letzten Prozeßmarke hat zusätzlich ein objektautonomes Schalten von e4 stattgefunden, wo-

bei im Prozeß wie erwünscht nicht unterschieden werden kann, ob dies während des Transports oder danach geschehen ist.

4.3.3 Diskussion

Netze in Netzen wurden in diesem Abschnitt auf zwei Weisen dargestellt, die beide gegenüber bisherigen Ansätzen erweiterte Modelle ermöglichen. Zum einen können auf Ebene der Objektnetze Feature Structures als gefärbte Marken eingesetzt werden, so daß eine Kombination aus aktiven Marken (Objektnetze in Systemnetzen) und passiven Marken (Feature Structures) möglich wird.

Die erste Darstellung, die System- und Objektnetz durch ein FSNet zusammenfaßt, bietet mehr Flexibilität bei der Struktur des Systemnetzes. Am behandelten Beispiel wurden einige der möglichen Erweiterungen noch nicht deutlich, die hier kurz skizziert werden sollen.

Folgendermaßen lassen sich mehrere Objektnetze in einem Systemnetz darstellen. Für jedes Objektnetz wird ein eindeutiger Typ definiert, wobei alle Objektnetz-Typen disjunkt sind. Die Prozeß-Marken erhalten ein weiteres Feature, daß auf einen Knoten von dem Typ verweist, der das zum Prozeß gehörende Objektnetz repräsentiert. Es können dann nur solche Prozesse unifiziert werden, die zu dem gleichen Objektnetz gehören. Außer dieser einfachen Einfärbung von Objektnetz-Prozessen ist weiterhin sogar denkbar, daß die Typen, welche die Objektnetze repräsentieren, in Subsumptionsrelation stehen dürfen. Diese müßte einhergehen mit einer entsprechenden Verhaltensvererbung zwischen den Netzstrukturen der Objektnetze, wie sie in [Aalst und Basten 1997, Aalst und Anyanwu 1999] vorgeschlagen wird. Damit wäre es möglich, Prozeßmarken unterschiedlicher Objektnetze zu unifizieren, wenn diese in einer Subsumptionsbeziehung stehen. Wie weit dies einem gewünschten Verhalten entspricht, muß noch untersucht werden.

Für einen Ansatz mit mehreren Objektnetzen schlägt Valk in [Valk 1999] eine zusätzliche Erweiterung vor, um Objektnetze direkt, also ohne Interaktion mit dem Systemnetz, interagieren zu lassen. Auch diese Variante ist als Erweiterung des ersten Ansatzes möglich, indem wie beim objektautonomen Schalten spezielle Systemnetztransitionen die direkte Objektinteraktion simulieren.

Als letzte Erweiterung stellt auch die Modifikation der Objektnetz-Struktur zur Laufzeit wie in [Farwer 2000b] mit diesem Ansatz kein Problem dar. Die Frage ist, ob solche Modifikationen überhaupt erlaubt sein sollten, da selbstmodifizierende Programme aufgrund ihres unvorhersehbaren Verhaltens in der Informatik verpönt sind. Im Bereich der Agenten (siehe Abschnitt 5.3.1), in dem eine hohe Flexibilität und Anpassungsfähigkeit verlangt wird, sind solche selbstmodifizierenden Ansätze aber durchaus gefragt. Ein formal fundiertes Modell wie die Linerarlogischen Petrinetze von Farwer oder die vorgestellten FSNets könnte den Nachteil der Unvorhersehbarkeit relativieren. Allerdings sind erfahrungsgemäß mit steigender Dynamik von Modellen um so mehr Probleme unentscheidbar. Beispielsweise ist die Erreichbarkeit

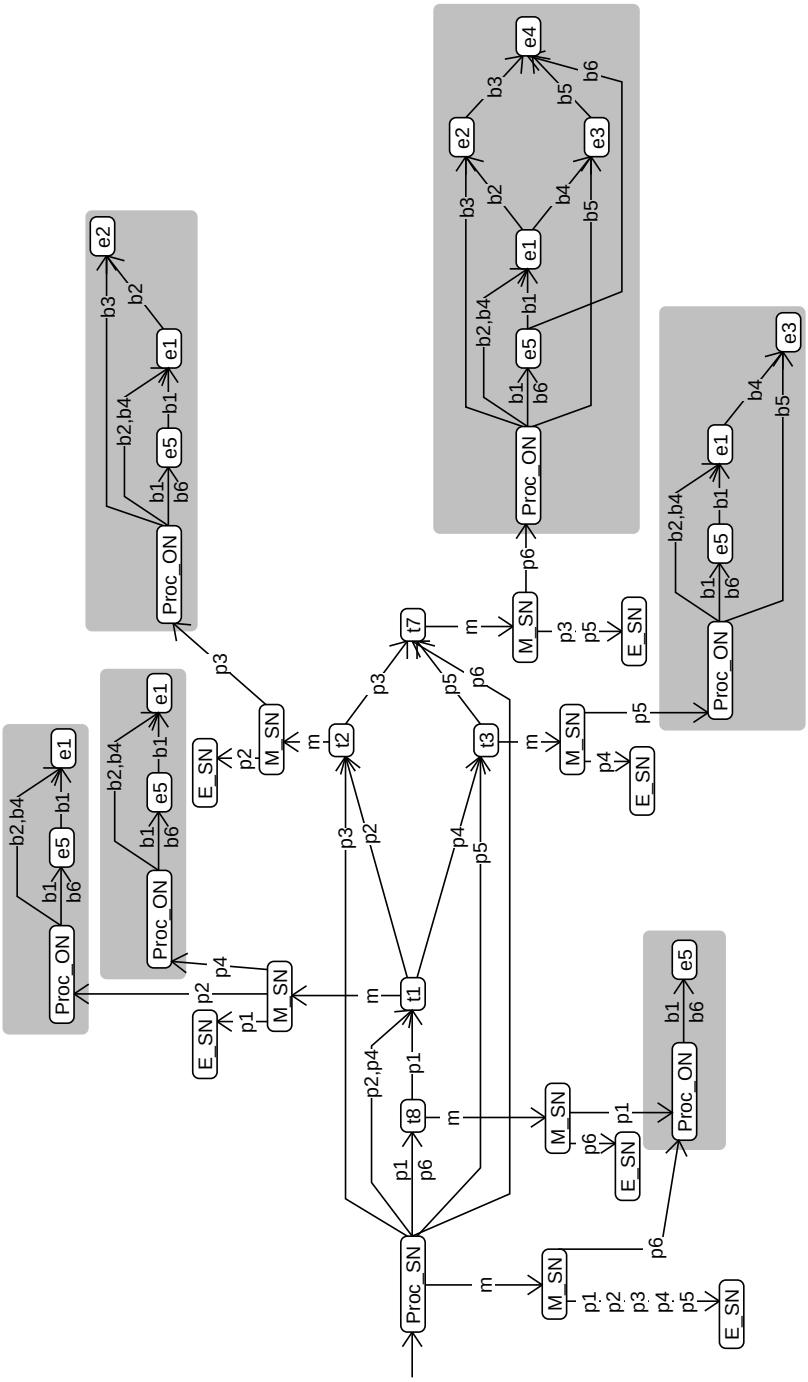

Abbildung 4.35: Feature-Structure-Prozeß des EOS aus Abbildung 3.19 über dem Typsystem aus Abbildung 4.33.

einer Markierung in S/T-Netzen entscheidbar; [Farwer 1999] zeigt aber, daß dieses Problem für Systemprozesse von EOS nicht entscheidbar ist.

Als ein Nachteil der Abbildung von EOS auf erweiterte FSNets ist zu nennen, daß die Objektnetze nicht aktive Objekte sind, sondern von ihrer Umgebung, dem Systemnetz, ausgeführt werden. Dies ist zwar nicht im eigentlichen Sinne der Netze in Netzen, dennoch aber in vielen Zusammenhängen eine angemessene Modellierung. Man denke an virtuelle Maschinen oder andere Interpreter, die ein als Daten gegebenes Programm Schritt für Schritt ausführen. Das Programm wird zwar konzeptuell als aktives Objekt angesehen, kann aber erst mit Hilfe einer entsprechenden Umgebung ausgeführt werden. Von diesem Sachverhalt soll andererseits in vielen Fällen abstrahiert werden.

Ein weiterer Nachteil ist, daß zur Konstruktion der Objektnetz-Prozesse der Aufruf einer Funktion nötig ist, was im Basisformalismus nicht vorgesehen ist. Damit reichen Basis-FSNets nicht aus, um elementare FSNets im Sinne von Netzen in Netzen zu simulieren, was letztendlich an der Bestimmung des maximalen S-Schnitts von unifizierten Prozessen scheitert (siehe oben). In Abschnitt 4.2.4 wurde gezeigt, daß jedes EFSNet sowohl nach Referenz- als auch nach Wertsemantik von einem universellen EFSNet simuliert werden kann. Die Konstruktion von EFSNet-Prozessen könnte durch ein EFSNet auf ähnliche Weise erfolgen, da die Anwendung der Prozeß-Transitionsregeln analog zur Anwendung der Transitionswirkungen erfolgt. Auch das Kopieren der Marken im Falle von Wertsemantik sollte mit einem ähnlichen Verfahren wie bei dem universellen EFSNet möglich sein. Das Problem liegt jedoch in der Konstruktion des maximalen S-Schnitts (Funktion scut), die durchgeführt werden muß, damit die Prozeß-Transitionsregeln anwendbar sind. In Abschnitt 4.3 wurde diskutiert, daß von entsprechend modifizierten Prozeß-Transitionsregeln an Stelle der Prozesse die maximalen S-Schnitte sukzessive konstruiert werden könnten. In der Wertsemantik muß aber der maximale S-Schnitt der Unifikation von Prozessen bestimmt werden, was sich so nicht darstellen läßt. Damit ist zu vermuten, daß ein EFSNet Netze in Netzen nur nach Referenzsemantik simulieren kann. Ein solches Netz wurde aber im Rahmen dieser Arbeit nicht konstruiert und ist damit Gegenstand zukünftiger Forschung.

Als letzter Nachteil des ersten Ansatzes zur Simulation von EOS mit FSNets läßt sich festhalten, daß durch die unterschiedlichen Modelle für System- und Objektnetz ein n-Ebenen-Modell (siehe Abschnitt 3.3.1) nicht auf natürliche Weise ergibt, wenn es wahrscheinlich auch ähnlich wie das objektautonome Schalten simulierbar wäre.

Der zweite Ansatz in Abschnitt 4.3.2 verleiht EFSNet-Objektsystemen als einer speziellen Variante der Netze in Netzen eine Prozeßsemantik. Dafür wird das Systemnetz auf ein elementares FSNet eingeschränkt. Trotz oder gerade mit Hilfe dieser Einschränkung behebt dieser Ansatz die Defizite des ersten Ansatzes: Die Objektnetze sind aktive Objekte, deren Schalten direkt in der Prozeßsemantik verankert ist. Die Prozeßsemantik greift nur auf die formal definierten Funktionen valmark und scut zurück, ohne daß diese aus einem FSNet heraus aufgerufen werden. Schließlich

ist es durch die gleichartige Darstellung von System- und Objektnetz möglich, ein Modell mit beliebig vielen Ebenen zu betrachten. Aufgrund der Ausführungen im vorherigen Abschnitt sollte deutlich geworden sein, wie ein solches n-Ebenen-Modell zu definieren wäre.

In dieser Arbeit wird damit erstmals eine Prozeßsemantik für gefärbte Netze in Netzen vorgestellt. Als besondere Eigenschaft kann diese Prozeßsemantik außerdem sowohl Wert- als auch Referenzsemantik darstellen.

Mit diesen Ergebnissen beenden wir den theoretischen Teil dieser Arbeit und wenden uns der Erweiterung des Modells in bezug auf die Modellierung praxisrelevanter Probleme zu.

4.4 Höhere Feature-Structure-Netze

Basis-FSNets und elementare FSNets stellen bewußt einfach gehaltene formale Modelle dar, die bestimmte Phänomene im Bereich der verteilten Systeme modellieren können (siehe Abschnitt 1.1). Die Familie der Feature-Structure-Netze wird in diesem Abschnitt um höhere FSNets ergänzt, die eine deutliche Ausrichtung auf den praktischen Einsatz bei der Modellierung zeigen.

Höhere FSNets bauen auf den Konzepten und Notationen der Basis-FSNets auf und erweitern diese um zusätzliche Konstrukte, die aus anderen Petrinetzformalismen entliehen werden. Dabei werden vor allem die in Abschnitt 3.3.3 vorgestellten Referenznetze herangezogen. Dies ist aus mehreren Gründen naheliegend.

Referenznetze sind formal definiert und erlauben eine objektorientierte Strukturierung von Petrinetzen, bieten aber im Bereich der Daten- bzw. Informationsmodellierung lediglich Tupel, Listen und eine Java-Schnittstelle. Damit ist zwar im Prinzip die Verwendung beliebiger Datentypen möglich, aber nur für einen Java-Programmierer oder durch Nutzen zusätzlicher Werkzeuge zugänglich.

Im Bereich der Modellierung von Information und Daten zeigen FSNets ihre Stärken. Zur Strukturierung komplexer Prozeßmodelle wurden allerdings in dieser Arbeit noch keine Vorschläge unterbreitet. Es liegt deshalb nahe, die Vorteile beider Ansätze zu kombinieren und die objektorientierte Prozeßmodellierung der Referenznetze mit der objektorientierten Datenmodellierung der FSNets zu vereinen.

Ein anderer Grund ist praktischer Natur: Wie in Abschnitt 4.5 dargestellt, baut das Werkzeug zur Unterstützung von FSNets auf das Referenznetzwerkzeug Renew auf. Diese Vorgehensweise wurde gewählt, da Renew über eine sehr offene Architektur verfügt und weiterhin der Autor durch seine Beteiligung an dem Projekt mit Renew bereits vertraut war. Eine Integration mit Konzepten, die in demselben Werkzeug ohnehin bereits realisiert sind, fällt offensichtlich leichter, als andere aus der Petrinetz-Theorie bekannte Strukturierungskonzepte heranzuziehen.

Zur Integration von Referenznetzen und FSNets sind verschiedene Ansätze denkbar. Zum einen könnten Referenznetze um Feature Structures als Marken erweitert

werden, zum anderen FSNets um die Konzepte der Referenznetze. Der erstgenannte Weg wurde eine Zeit lang begangen und soll hier kurz skizziert werden, um anschließend die aus dem zweiten Weg resultierenden höheren FSNets ausführlicher vorzustellen.

Referenznetze, so wie sie im Werkzeug Renew realisiert wurden, können als Marken beliebige (Java-)Objekte enthalten. In Abschnitt 2.4 wurde eine in Java implementierte Feature-Structure-Bibliothek vorgestellt. Diese kann direkt unter Renew eingesetzt werden, wodurch Feature Structures als Marken in Referenznetzen zur Verfügung stehen. FSNets unterscheiden sich aber in einigen Aspekten von Referenznetzen, so daß eine spezielle Unterstützung nötig ist.

Statt dieser Spezialisierung sollen im folgenden *höhere Feature-Structure-Netze* (höhere FSNets, *high-level FSNets* oder HFSNets) vorgestellt werden. Die Erweiterung um die Konzepte Netzexemplar, Referenz und synchroner Kanal werden in Abschnitt 4.4.1 beschrieben. Eine formale Definition von HFSNets erfolgt hier nicht, da sie den Rahmen des formalen Teils dieser Arbeit sprengen würde. Außerdem steht die vollständige formale Definition von Referenznetzen selbst noch aus, die in [Kummer 2001] erscheinen wird. Neben den wichtigsten, oben genannten Konzepten werden die in Abschnitt 3.2.4 angesprochenen erweiterten Kantenarten auf FSNets übertragen (Abschnitt 4.4.2).

4.4.1 Netzexemplare und synchrone Kanäle in HFSNets

Ein Referenznetz stellt, wie in Abschnitt 3.3.3 beschrieben, ein Muster für konkrete Netzexemplare dar. Dieses Konzept kann direkt in FSNets übernommen werden. Damit verschiedene Netzexemplare miteinander kommunizieren können, müssen diese über Referenzen und synchrone Kanäle aufeinander zugreifen können. Bei der Übertragung dieser Konzepte muß man etwas sorgfältiger vorgehen.

In Abschnitt 2.4.2 wird als praxisorientierte Erweiterung von Feature Structures eingeführt, daß beliebige Java-Typen in Typsystemen auftreten und die entsprechenden Attribute in Feature Structures auf (als Feature Structures dargestellte) Java-Objekte verweisen dürfen. Netzexemplare werden in Renew durch Exemplare der Java-Klasse `de.renew.simulator.NetInstanceImpl` dargestellt. Für das Erzeugen neuer Netzexemplare gibt es eine besondere Syntax (siehe Abschnitt 3.5), die in HFSNets dank der Feature-Structure-Erweiterungen nicht benötigt wird. Neue Netzexemplare können als Java-Objekte erzeugt und beispielsweise als Attributwert in eine Feature Structure eingetragen werden. In der Darstellung werden Netzexemplare als Exemplartypen der Klasse `de.renew.formalsm.fsnet.FSNetInstance` dargestellt, die eine spezielle Subklasse von `NetInstanceImpl` darstellt. Der Name des zu instantiierenden Netzes wird unter dem Feature `of` angegeben. Ausdrücke dieser Art können sowohl als Anfangsmarkierungen als auch als Transitions- oder Kantenanschriften vorkommen. Java-Objekte werden in HFSNets immer erst instantiiert (siehe Abschnitt 2.4.2), nachdem das Netzexemplar zu dem Netz, in dem sich die

Java-Feature-Structures als Anfangsmarkierung befinden, erzeugt wurde, bzw. nachdem die Transition, in deren Anschriften die Java-Feature-Structurevorkommen, geschaltet hat.

Um einen synchronen Kanal aufzurufen, kann auf eine in einer Feature Structure enthaltene Netzreferenz über eine entsprechende Knotenreferenz zugegriffen werden. Der Punkt, in dem sich Renew-Netze und FSNets unterscheiden, ist der Austausch von Information über den synchronen Kanal. In Renew findet an dieser Stelle ein Vergleich des Kanalnamens und eine parameterweise Unifikation statt. Allerdings ist diese Unifikation einfacher als bei Feature Structures, da sie mit der Unifikation in der Prädikatenlogik vergleichbar ist. Es handelt sich also um eine Term-, nicht wie bei Feature Structures um eine Graphunifikation. Diese beiden Unifikationsarten sollten nicht in einem Formalismus vermischt werden, so daß für synchrone Kanäle in FSNets auch eine Graphunifikation verwendet wird. Auch stellen Feature Structures bereits komplexere Datenstrukturen dar, so daß die Angabe eines Kanalnamens und einer Parameterliste überflüssig ist. Der Name wird konzeptuell durch den Typ des Wurzelknotens des Kanalparameters ersetzt. So können die in Referenznetzen benutzten Kanalnamen durch entsprechende Typen dargestellt werden, wobei durch Subsumption zwischen Typen als Erweiterung beispielsweise Zielkanäle möglich sind, die einen allgemeineren Typ angeben und damit auf verschiedene speziellere Anfragen reagieren. Dies ähnelt der Verarbeitung von Ausnahmen (*exceptions*) in Java (siehe [Gosling et al. 1996]). Mit den in Abschnitt 2.3.5 eingeführten Tupeln und Listen sind auch unbenannte Kanäle einfach darstellbar.

Es bleibt für einen synchronen Kanal die Unterscheidung zwischen Start- und Zielkanal, die beide eine Feature Structure spezifizieren, wobei der Startkanal zusätzlich eine Referenz auf das Netzexemplar angibt, mit dem eine Synchronisation stattfinden soll. Wie bei Kanten in FSNets werden die Parameter des synchronen Kanals innerhalb der Transitionsregel dargestellt. Der Start- oder Zielkanal erhält einen Pfad, der auf die entsprechende Stelle der Transitionsregel verweist.

Damit ergibt sich wie bei Referenznetzen gegenüber gefärbten Netzen eine erweiterte Aktivierungs- und Schaltregel. Beide beruhen in FSNets auf dem Schaltmodus (siehe Definition 4.4), so daß ein erweiterter Schaltmodus definiert werden muß. Ein solcher erweiterter Schaltmodus muß das Schalten mehrerer synchronisierter Transitionen beschreiben. Jede der Transitionen besitzt ihre eigene Markenbindung (siehe Definition 4.3). Die Markenbindungen müssen paarweise disjunkt sein, d.h. keine Marke darf (von verschiedenen Transitionen) mehrfach konsumiert werden.

Eine Transition t sei in einem HFSNet aktiviert, wenn es einen erweiterten Schaltmodus gibt, der einen gültigen Schaltmodus für alle synchron schaltenden Transitionen darstellt.

(a) Es gibt einen Schaltmodus sm zu $EFSN$, t und m_1.

(b) Für jeden Startkanal $x_i : \pi_i$ der Transition t gibt es im Netzexemplar x_i eine Transition t_i mit einem Zielkanal : π'_i, die lokal in einem Schaltmodus sm'_i

aktiviert ist und es gilt: $share := sm@\pi \sqcup \bigsqcup\{sm'_i@\pi'_i\}$ ist definiert.

Die Folgemarkierung ergibt sich durch das gleichzeitige Schalten aller synchronisierten Transitionen, wobei durch den erweiterten Schaltmodus Information ausgetauscht werden kann.

Anwendungsbeispiele, in denen die Referenznetz-Erweiterungen von FSNets verwendet werden, finden sich in Kapitel 5.

4.4.2 Test-, Reservierungs- und Inhibitorkanten in HFSNets

In diesem Abschnitt werden kurz die Besonderheiten der in Abschnitt 3.2.4 vorgestellten erweiterten Kantenarten in bezug auf FSNets beschrieben.

Auf die Bedeutung von Testkanten in FSNets wurde bereits in Abschnitt 4.1.3 hingewiesen: Durch die Unifikationssemantik von FSNets ist es in Basis-FSNets nicht ohne weiteres möglich, den Inhalt eine Marke zu lesen, ohne diesen zu verändern. Vor allem bei der Anwendung von Regeln, die sich mit Feature Structures sehr gut darstellen lassen, ist eine Modifikation der Regel-Feature-Structure aber unerwünscht.

Die Semantik einer Testkante in HFSNets ergibt sich wie folgt. Die Bindung von Marken an Kanten entspricht der in Abschnitt 3.2.4 gegebenen Semantik: Eine Marke darf durch beliebig viele Testkanten gleichzeitig getestet werden, aber nicht durch eine synchron schaltende Transition konsumiert werden.

Test- und Reservierungskanten weichen in bezug auf den Markeninhalt aber von der Kombination aus gleich beschrifteter Ein- und Ausgangskante ab. Wie in Abschnitt 4.1.3 bemerkt wurde, transportieren gleich beschriftete Ein- und Ausgangskanten in Basis-FSNets nicht unbedingt eine Marke mit demselben Wert, sondern der Wert kann in der Ausgangsmarke durch Unifikation spezialisiert worden sein. Dieses gegenüber gefärbten Netzen veränderte Verhalten soll für Test- und Reservierungskanten dem Standard-Verhalten entsprechen. Test- und Reservierungskanten lesen den Wert der an sie gebundenen Marke, ändern diesen aber nicht.

In der formalen Semantik ließe sich dieses Verhalten hinzufügen, indem Test- und Reservierungskanten bei der Markenbindung wie in Abschnitt 3.2.4 beschrieben eingeführt werden, mit der Transitionsregel unifiziert werden, auf die Folgemarkierung aber keinen weiteren Einfluß haben.

Eine weitere in Abschnitt 3.2.4 vorgestellte Kantenart ist die Inhibitorkante. Diese testet in gefärbten Netzen, ob eine Stelle *keine* Marke enthält, die den Wert trägt, der sich aus der Kantenanschrift ergibt. Übertragen auf FSNets wird konsequenterweise nicht die Gleichheit des Feature-Structure-Ausdrucks an der Inhibitorkante mit einer Feature-Structure-Marke in der Stelle gefordert, um die Transition zu deaktivieren, sondern die Unifizierbarkeit. Eine Transition t mit einer Inhibitorkante zur Stelle s ist demnach genau dann in einem Schaltmodus aktiviert, wenn es in s keine Marke gibt, die mit dem Pfadwert des Schaltmodus unter dem an der Inhibitorkante genannten Pfad unifizierbar ist. Dies ist als Sonderfall wie bei gefärbten Netzen unter anderem dann der Fall, wenn die Stelle unmarkiert ist.

4.5 Werkzeugunterstützung: FSRenew

Um die Erstellung und Ausführung von höheren FSNets durch ein Software-Werkzeug zu unterstützen, wurde auf das für Referenznetze entworfene Petrinetz-Werkzeug Renew (siehe Abschnitt 3.5) aufgebaut.

Renew bietet eine offene Architektur für die Bearbeitung und Simulation von höheren Petrinetzen. Der spezielle Formalismus der Referenznetze wurde als ein *Modus* von Renew realisiert, der als Standard angenommen wird. Beim Start des Werkzeugs können andere Modi, wie zum Beispiel der in Abschnitt 4.5 beschriebene FSNet-Modus, über einen Parameter angegeben werden. Durch die konsequente Trennung von graphischer Benutzungsoberfläche und dem unterliegenden Petrinetzmodell sowie dem Simulator kann das Werkzeug mit einem entsprechenden Programmieraufwand für beliebige Petrinetzformalismen angepaßt werden.

Im Rahmen dieser Arbeit wurde Renew um einen Modus für höhere FSNets erweitert. Das so erweiterte Werkzeug soll im folgenden FSRenew genannt werden.

Auf die Einzelheiten der Architektur und der Bedienung von FSRenew soll hier nicht eingegangen werden. Diese werden vielmehr mit der entsprechenden Renew-Version in Form eines Benutzerhandbuchs und technischer Dokumentation zur Verfügung gestellt. An dieser Stelle wird ein kurzer Überblick über die Möglichkeiten von FSRenew gegeben.

Auf Seiten der graphischen Bearbeitung von Modellen mußte Renew um eine Typmodellierungs-Komponente erweitert werden. Die Typmodellierung bei Referenznetzen findet normalerweise außerhalb des Werkzeugs durch Java-Programmierung statt. In FSRenew ist eine wie in Abschnitt 2.1.1 dargestellt eingeschränkte Modellierung von UML-Klassendiagrammen möglich, aus denen ein Konzept- und damit ein Typsystem generiert wird (siehe Abschnitt 2.1.3). Über diesem Typsystem können nun Feature Structures als Anschriften in den wie mit Renew erstellten Netzen verwendet werden. Feature Structures werden in einer speziellen, aber einfach zu erlernenden Syntax eingegeben. Ein mit `JavaCC` ([Sun 2000b]) implementierter Parser übersetzt die Eingaben in Objekte der in Abschnitt 2.4 vorgestellten Feature-Structure-Bibliothek. Dabei werden syntaktische und semantische Fehler dem Benutzer mit Beschreibungen und genauen Angaben zur Fehlerposition gemeldet. Syntaktisch korrekte Feature Structures werden von FSRenew graphisch dargestellt. Dabei kann sowohl auf die in Abschnitt 2.3.1 eingeführte AVM-Notation als auch auf die in Abschnitt 2.3.5 eingeführte UML-Notation zurückgegriffen werden, wobei zwischen beiden Darstellungen umgeschaltet werden kann. Die ebenfalls in Abschnitt 2.3.5 vorgestellte Listennotation wird in der graphischen Ausgabe unterstützt.

Während der Simulation werden auch berechnete Feature Structures in die graphische Darstellung übersetzt. Um große Strukturen übersichtlich zu halten, wurden zwei spezielle Bedienelemente eingeführt. Schließ-Bedienelemente dienen dazu, Unterstrukturen einer Feature Structure zu verbergen oder wieder anzuzeigen.

Knotenreferenzen können selektiert werden, um alle Vorkommen einer Knotenreferenz aufzudecken und hervorgehoben darzustellen. Diese erweiterten Eigenschaften können auch für die Detaildarstellung von Java-Objekten genutzt werden und werden in dieser Funktion bereits im aktuellen Renew-Benutzerhandbuch beschrieben ([Kummer et al. 2000b]).

Sämtliche aus Renew bekannten Funktionen wie das Öffnen von Netzexemplaren in weiteren Fenstern und die unterschiedlichen Darstellungsmodi für Markierungen bleiben in FSRenew erhalten, so daß ein mächtiges Modellierungswerkzeug für FSNets zur Verfügung steht.

Die Vorteile von FSRenew gegenüber Renew sind im folgenden nochmals stichpunktartig zusammengefaßt.

- Das UML-basiertes Typmodellierungswerkzeug kann zur informationsorientierten Modellierung auch von Benutzern ohne Programmierkenntnisse eingesetzt werden.
- Die Eingabe von Feature Structures wird durch detaillierte Syntax- und Typfehlermeldungen unterstützt.
- Komplexe Substrukturen von Feature Structures können bei der Beschriftung von Netzen und während der Simulation verborgen werden.
- Durch Exemplartypen können Java-Objektgraphen auf einfache Weise erzeugt werden. Damit ist beispielsweise die Erstellung einfacher GUIs ohne Programmierkenntnisse möglich.

Sämtliche in dieser Arbeit gezeigten Abbildungen wurden mit FSRenew erstellt, was auch die Stärken des Werkzeugs als graphischer Editor zeigt. Die im folgenden Kapitel gezeigten Modellierungsbeispiele sind alle mit FSRenew erstellt worden und können in der angegebenen Form ausgeführt werden. Damit wird ein Werkzeug für FSNets zur Verfügung gestellt, das nicht nur zur Modellierung, sondern auch zum Prototyping verteilter Systeme geeignet ist.

4.6 Diskussion

Als Abschluß dieses Kapitels sollen die erzielten Eigenschaften von FSNets zusammengefaßt werden, die anhand der Modellierungsbeispiele im folgenden Kapitel belegt werden sollen.

Mit FSNets wird eine graphische, ausführbare, sprachunabhängige und intuitive Modellierungstechnik für verteilte Systeme bereitgestellt. Die genannten Eigenschaften treffen auf die beiden Teiltechniken Feature Structures und Petrinetze zu. Bei der Kombination zu einer gemeinsamen Technik wurde darauf geachtet, daß die positiven Eigenschaften der Einzeltechniken erhalten bleiben und sich zu einer konsistenten Technik ergänzen.

Graphisch. Die Anschaulichkeit und Verständlichkeit graphischer Repräsentationen von Modellen ist vor allem deshalb von erheblicher Bedeutung, weil oft Domänenexperten in die Modellierung einbezogen werden, die keine Modellierungsexperten sind ([Jablonski et al. 1997], S. 162). Sowohl Petrinetze als auch Feature Structures besitzen eine graphische Repräsentation. Falls die Darstellung von Feature Structures als Attribut-Wert-Matrix (AVM) oder als UML-Objektsicht nicht hinreichend graphisch erscheint, kann die Graphnotation verwendet werden (siehe Abschnitt 2.3). Wichtig für die angebliche „Selbsterklärungsfähigkeit graphischer Repräsentationen“ ([Jablonski et al. 1997], S. 162) ist, daß sich graphische Notation ebenso wie textuelle an bekannte Konventionen halten müssen, um eindeutig interpretierbar zu sein. In [Blackwell 1996] werden die Vor- und Nachteile graphischer Notationen kritisch betrachtet. Die hier vorgeschlagenen Notationen richten sich nach den Konventionen und Standards für Petrinetze und UML, so daß die Voraussetzungen für ein allgemeines und einfacheres Verständnis von graphischen FSNet-Modellen gegeben sind.

Ausführbar. Modelle dienen der Veranschaulichung und Konzeptualisierung von Domänen oder Problemstellungen. In [Moldt 1996], [Kennedy 1999] und [Boger et al. 2000] wird argumentiert, daß ausführbare Modelle den Nutzen eines Modells erhöhen und Modellierungsfehler vermeiden helfen. Petrinetze sind für ihre formale operationale Semantik bekannt ([Jensen 1992]). Bei höheren Petrinetzen hängt die Ausführbarkeit von der Wahl der Daten- und Operationsmodellierung ab. Werden hier informale Techniken eingesetzt (beispielsweise natürliche Sprache), ist keine automatische Ausführung möglich. Feature Structures stellen zusammen mit den Operationen der Unifikation und der Pfadwert-Bildung dagegen einen formal definierten und, wie in Abschnitt 2.4 gezeigt wurde, implementierbaren Datentyp dar. Basis-FSNets werden in dieser Arbeit mitsamt einer operationalen Semantik definiert. Daß auch die praxisorientierte Erweiterung zu höheren FSNets ausführbare Modelle hervorbringt, wird durch die Realisierung eines entsprechenden Werkzeugs gezeigt.

Sprachunabhängig Bei der Modellierung von Systemen sollen frühestens im *Entwurf* Entscheidungen getroffen werden, die von der verwendeten Programmiersprache abhängen. Obwohl FSNets ausführbar sind, verwenden sie zur Beschreibung von Daten und Aktivitäten keine Programmiersprache. Petrinetze stellen eine mathematische bzw. graphische Spezifikation von Abläufen oder *Steuerfluß* dar, die nicht durch konkrete Sprachkonstrukte wie Schlüsselwörter für Schleifen, Verzweigungen und ähnliches angegeben werden muß. Feature Structures basieren in der hier verwandten Sichtweise auf Konzepten der Objektorientierung, orientieren sich aber in Syntax und Semantik nicht an einer konkreten Programmiersprache, sondern an UML. Der einzige Bezug zur konkreten Programmiersprache Java besteht in

der hier eingeführten Integration von Feature Structures und Java-Objekten, die eine optionale Eigenschaft der Werkzeugunterstützung darstellt. Werden im weiteren Sinne auch UML und Petrinetze als Sprachen betrachtet, sind FSNets nicht sprachunabhängig. Diese Sprachen befinden sich aber auf einer höheren Abstraktionsebene als die mit der Eigenschaft „sprachunabhängig“ gemeinten Programmiersprachen.

Intuitiv. Diese Eigenschaft wird hier in dem Sinne verstanden, daß die Modellierungstechnik auf einer überschaubaren Anzahl klar abgegrenzter Konzepte besteht. Dies ist sowohl bei Petrinetzen als auch bei Feature Structures der Fall. Während Petrinetze im wesentlichen auf den Konzepten Stelle und Transition basieren, werden komplexe Datenstrukturen in Feature Structures durch Typen und Attribute beschrieben. FSNets verwenden Feature Structures sowohl als Kanten-, Stellen und Transitionsanschriften als auch zur Darstellung der Marken, so daß das benötigte syntaktische Repertoire sehr viel kleiner ist als in anderen höheren Petrinetzformalismen. Die Trennung in prozeß- und informationsorientierte Aspekte sorgt für eine klare Abgrenzung der Techniken voneinander. Zusammen mit der graphischen Repräsentation und der Möglichkeit, Modelle „ausprobieren“ zu können, können FSNets mit Recht als intuitive Modellierungstechnik bezeichnet werden.

Kapitel 5

Anwendungsbeispiele für die Modellierung mit Feature-Structure-Netzen

Im vorherigen Kapitel wurde mit FSNets ein neuer Petrinetzformalismus definiert und untersucht. In diesem Kapitel soll anhand konkreter Beispiele aufgezeigt werden, daß sich FSNets tatsächlich als Modellierungstechnik für verteilte Systeme einsetzen lassen.

Statt einer einzelnen umfangreichen Fallstudie wird hier der Ansatz gewählt, eine gewisse Breite von Anwendungsgebieten abzudecken. Die Modellierung mit FSNets wird sowohl anhand allgemeiner Vergleiche mit den in den Kapiteln 2 und 3 vorgestellten Modellierungstechniken als auch durch konkrete Beispiele motiviert. Als repräsentative Anwendungsgebiete der verteilten Systeme wurden Problemstellungen aus den Bereichen Geschäftsprozeßmodellierung, E-Commerce und intelligente Agenten gewählt.

In Abschnitt 5.1 wird die Geschäftsprozeßmodellierung mit FSNets anhand von zwei Beispielen vorgestellt: einem Geschäftsprozeß aus der juristischen Verwaltung und einer Online-Bestellungsverarbeitung. Abschnitt 5.2 stellt Cosmos als eine Infrastruktur für elektronische Dienstemärkte vor und betrachtet die Ausführung elektronischer Verträge als FSNet-Modelle. In Abschnitt 5.3 wird schließlich eine allgemeine Architektur für BDI-Agenten sowie ein konkretes Beispiel für einen planenden und reaktiven Abfallsammler-Agenten durch FSNets modelliert.

5.1 Geschäftsprozeßmodellierung

Daß sich Petrinetze, wie sie auch FSNets zugrunde liegen, sehr gut zur Modellierung der Prozeßaspekte in Workflows eignen, wurde bereits in Abschnitt 3.4.2 belegt. FSNets werden in dieser Arbeit als Modellierungstechnik für Prozesse *und* Daten

bzw. Information ausgewiesen. Anhand der Workflow-Modellierung soll nun gezeigt werden, daß sie dieses Versprechen einlösen können. Wenden wir uns deshalb den informationsorientierten Modellierungsaspekten von Geschäftsprozessen zu, die in Abschnitt 3.4 genannt wurden. Dieses sind im wesentlichen Rollen, Aufgaben und Ressourcen.

Die Rollen der Workflow-Teilnehmer sind Teil der Informationsmodellierung und stehen im allgemeinen in Generalisierungsbeziehungen. Dabei ist zu beachten, daß Generalisierungen zwischen Rollen nach dem Substitutionalitätsprinzip der Objektorientierung zu lesen sind, nicht als Hierarchien. Während beispielsweise ein Abteilungsleiter in der Unternehmenshierarchie über dem Mitarbeiter steht, stellt sich die Generalisierung zwischen den entsprechenden Rollen entgegengesetzt dar: Die Befugnisse des Abteilungsleiters *erweitern* die des Mitarbeiters. Eine Generalisierung zwischen zwei Rollen ist also nicht als „Ein A ist ein B“, sondern als „Ein A kann sich verhalten wie ein B“ zu lesen. Die Generalisierung zwischen Personen und Rollen entspricht dagegen wieder der Intuition und zeigt von der Person zur Rolle. UML sieht zur Unterscheidung für den letzteren Fall das Stereotyp «`roleOf`» vor ([Hitz und Kappel 1999]).

Rollen können weiterhin in Assoziationen zu Daten oder Dokumenten stehen und Attribute zur näheren Klassifizierung tragen (Engineer, skill=3). Methoden werden in der Analysephase üblicherweise nicht betrachtet, so daß die in dieser Arbeit verwendete Typmodellierung alle zur Spezifikation einer Rollenhierarchie benötigten Techniken zur Verfügung stellt.

Der wichtigste Bereich der Informationsmodellierung in Geschäftsprozessen ist die Repräsentation von Aufgaben (*tasks*). In dem in Abschnitt 3.4 erwähnten Standard der WFMC wird einer Aufgabe eine Rolle zugeordnet, die für deren Ausführung verantwortlich ist. Auch Aufgaben können durch Angabe von Attributen genauer spezifiziert werden.

Ressourcen in Geschäftsprozessen sind die bearbeiteten Dokumente, Gegenstände etc. sowie die zur Bearbeitung benötigten Hilfsmittel. Auch Ressourcen können in verschiedene (Sub-)Typen unterteilt und mit Attributen versehen werden, um sie gezielt bestimmten Aufgaben zuzuteilen. Daten wie z.B. Attribute von Dokumenten werden unterteilt in solche, die den Ablauf des Geschäftsprozesses beeinflussen (Steuerinformation, *control data*) und andere, die nur innerhalb der Applikationen von Bedeutung sind (Applikationsdaten, *application data*). Zumindest Steuerinformation sollte im Workflow modelliert werden können, in einem integrierten System auch Applikationsdaten. Dies ist dann nötig, wenn die Semantik der Operationen, die Aufgaben realisieren, im Modell spezifiziert werden soll.

Schließlich kann ein Workflow-Exemplar selbst individuelle Attribute besitzen, die beispielsweise dessen Priorität steuern und damit eine Meta-Information des Workflows darstellen, die von der Workflow-Engine ausgewertet wird.

Die verschiedenen Workflow-Modellierungen in den folgenden Beispielen decken viele dieser Informationsaspekte ab. In Abschnitt 5.1.1 wird ein generisches Typ-

modell für Workflows aufgestellt und angewandt, das in Abschnitt 5.1.2 um die Behandlung strukturierter Dokumente erweitert wird. Es wird dargestellt, wie FSNet-Modelle sowohl zur Simulation als auch zur Inbetriebnahme (*enactment*) von Workflows verwendet werden können.

5.1.1 Ein Geschäftsprozeß einer Justizbehörde

Um mit einem einfachen Beispiel für die Modellierung von Geschäftsprozessen mit FSNets zu beginnen, ziehen wir das in [Valk 1999] verwendete Beispiel eines Geschäftsprozesses aus der niederländischen Justizbehörde heran.

Das Beispiel wird in [Valk 1999] folgendermaßen eingeführt: Nachdem sich eine Straftat ereignet hat und die Polizei einen Verdächtigen ermittelt hat, legt ein Sachbearbeiter eine Akte an. Die Akte wird als Kopie an einen Sekretär und einen weiteren Sachbearbeiter versandt. Während der Sekretär die Information in der Akte überprüft, ergänzt der zweite Sachbearbeiter die Akte um Hintergrundinformation über den Verdächtigen und bestimmte Daten der lokalen Verwaltung. Nachdem diese Aktivitäten abgeschlossen sind, wird der Fall von einem weiteren Sachbearbeiter untersucht und einem Staatsanwalt vorgelegt. Der Staatsanwalt entscheidet, ob der Verdächtige vor einem Tribunal angeklagt oder vorgeladen wird, oder aber der Fall abgewiesen wird.

Die Zuordnung von Aktivitäten geschieht bei Valk durch eine Modellierung der Rollen und ihrer „Kommunikationsverbindungen“ als Systemnetz und der Aufgaben und ihrer kausalen Abhängigkeiten als Objektnetz (siehe Abschnitte 3.3.1 und 3.4.2). Hier sollen dagegen wie oben bereits eingeleitet die Rollen und die Geschäftsobjekte, im Beispiel die Akte, durch Typen und Feature Structures modelliert werden. Aufgaben werden durch Transitionen beschrieben, deren Transitionsregeln die Zuordnung zu Rollen und die Veränderung des Zustands von Geschäftsobjekten in Form von Feature Structures spezifizieren.

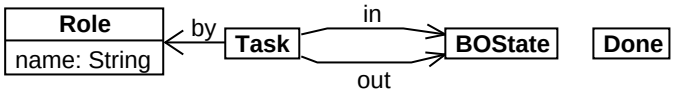

Abbildung 5.1: Ein einfaches generisches Typsystem zur Workflow-Modellierung.

Diese Art der Modellierung wird zunächst als generisches Schema (Meta-Modell) eingeführt. Abbildung 5.1 zeigt das allgemeine (und recht triviale) Typmodell, welches im folgenden für die konkrete Geschäftsprozeß- und Workflowmodellierung herangezogen wird. Jede Aufgabe (Typ Task) bekommt eine Rolle (Typ Role) und zwei Geschäftsobjekt-Zustände (*business object state*, Typ BOState) zugeordnet, wobei das Feature in den Zustand vor und das Feature out den Zustand nach Bearbeitung

der Aufgabe repräsentiert. Das Feature name der Rolle ist nicht etwa der Name einer konkreten Rolle (diese werden wie unten gezeigt als Subtypen modelliert), sondern das Namensattribut eines Workflowteilnehmers, der diese Rolle einnimmt. Done dient als Typ des synchronen Kanals (siehe Abschnitt 4.4), der die Beendigung des Workflows anzeigt.

Für dieses einfache Modell sind alternative Ansätze denkbar. Beispielsweise muß in einem Workflow-System die Wirkung auf des Zustand von Geschäftsobjekten nicht unbedingt berücksichtigt werden, wenn sie den Anwendern oder externen Programmen überlassen wird. Denkbar wäre auch eine Modellierung des gesamten Geschäftsobjekts, nicht nur seines Zustands, die im Beispiel in Abschnitt 5.2 vorkommt.

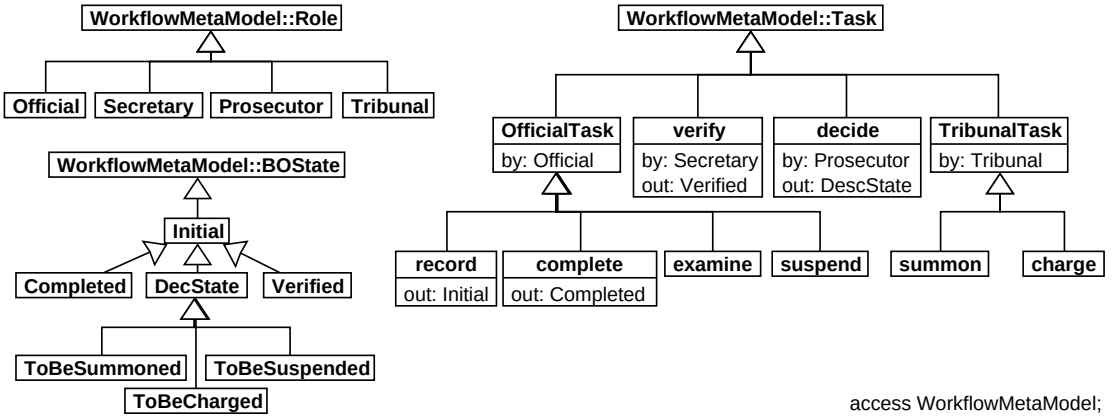

Abbildung 5.2: Das Typmodell für den Geschäftsprozeß des niederländischen Gerichts.

Abbildung 5.2 zeigt das Typmodell für den oben beschriebenen Geschäftsprozeß des niederländischen Gerichts. Von jedem der drei Typen des Workflow-Meta-Modells Role, Task und BOState werden für die Anwendungsdomäne spezielle Subtypen gebildet.

Als Rollen stehen der Sachbearbeiter (Official), der Sekretär (Secretary), der Staatsanwalt (Prosecutor) und das Tribunal zur Verfügung. Die Rollen sind hier als *disjunkte* Subtypen (siehe Abschnitt 2.1.1) von Role modelliert, was eine Zuordnung eines Teilnehmers zu mehreren dieser Rollen nicht erlaubt. Wo es inhaltlich sinnvoll ist, könnte hier stattdessen eine *überlappende* Vererbung spezifiziert werden. Dabei ist aber zu beachten, daß sowohl in der Schaltregel von Basis-FSNets (Abschnitt 4.1.3) als auch in der Erweiterung um synchrone Kanäle (Abschnitt 4.4.1) nur eine Unifizierbarkeit, nicht eine Subsumption der Marken, Parameter und der Transitionsregel verlangt wird. Ohne Subsumptionsbeschränkungen, die man in HFSNets als Wächter einführen könnte, kann bei überlappenden Subtypen nicht ausgedrückt werden, daß ein Knoten zu einem bestimmten Subtyp gehören *muß*. Deshalb wird hier keine überlappende Vererbung betrachtet.

Die verschiedenen Zustände der Akte sind als Subtypen von BOState modelliert. Da es sich um einen sehr einfachen Lebenszyklus handelt, bietet sich eine solche Modellierung in diesem Fall an. Bei komplexeren Zuständen eines Geschäftsobjekts sollte hierfür nach dem Ansatz der Netze in Netzen (siehe Abschnitt 3.3) das Zustandsmodell wiederum als Petrinetz modelliert werden und nur die Daten des Geschäftsobjekts durch Typen. Die Zustände der Akte entsprechen im wesentlichen den Ergebnissen der Aktivitäten, wobei durch die Subtypen von DecState (*decision state*) die verschiedenen Entscheidungen des Staatsanwalts modelliert werden.

Die für Rollen und Zustände von Geschäftsobjekten definierten Typen können nun für die Beschreibung der Aufgaben eingesetzt werden. Für jede in der Beschreibung des Geschäftsprozesses erwähnte Ausgabe gibt es einen Subtyp von Task, der hier ausnahmsweise klein geschrieben wurde, um den Verbalcharakter von Aufgaben deutlich zu machen. Die „abstrakten“ Aufgaben-Typen OfficialTask und TribunalTask dienen der Zusammenfassung gemeinsamer Features der entsprechenden Aufgaben. Die Aufgaben spezialisieren den Typ ihrer Features by, in und out entsprechend der oben angegebenen Beschreibungen dieser Features im Workflow-Meta-Modell. Beispielsweise verlangt die Aufgabe verify, von jemandem mit der Rolle „Sekretär“ ausgeführt zu werden, und daß sich das bearbeitete Geschäftsobjekt nachher im Zustand „überprüft“ befindet.

Durch die Verlagerung der Zuweisung von Rollen und Objektzuständen wird die Spezifikation des eigentlichen Workflows als FSNet einfacher und die verschiedenen Sichten können während der Modellierung besser getrennt entworfen werden. Für jede Transition muß angegeben werden, welche Aufgabe mit dieser assoziiert sein soll. Die Wahl der Rolle und die Veränderung des Objektzustands ergibt sich aus dem Typ der Aufgabe. Nur die Bindung des Geschäftsobjekt-Zustands an eine Marke aus dem Netz muß noch explizit gemacht werden.

Abbildung 5.3 zeigt den Gerichts-Geschäftsprozeß als HFSNet. Zur Modellierung werden hier die in Abschnitt 4.4 eingeführten HFSNets genutzt, die durch die Kombination mit Referenznetzen die Konzepte des Netzexemplars, der Referenz und des synchronen Kanals zur Verfügung stellen. Diese erweisen sich bei der Modellierung von Workflows als sehr nützlich.

Der Doppelpunkt vor den Feature Structures, welche die Aufgaben der Transitionen beschreiben, steht für einen Zielkanal (siehe Abschnitte 4.4.1 und 3.3.3). Damit kann das Schalten einer Aufgaben-Transition durch ein externes Ereignis mit einer entsprechenden Feature Structure als Parameter ausgelöst werden. Dies kann entweder durch ein anderes FSNet geschehen (siehe unten) oder durch eine externe Steuerung. Letzteres könnte durch die Workflow-Erweiterungen für Renew (siehe Abschnitt 3.5.2) realisiert werden, so daß der modellierte Geschäftsprozeß direkt als Workflow ausgeführt und über den Renew-Workflow-Client bedient würde. Die Renew-Workflow-Erweiterungen konnten aus zeitlichen Gründen nicht mehr in dieser Weise für FSNets angepaßt werden, dies ist aber für die Zukunft geplant.

Die Bedeutung von Referenzen und Netzexemplaren wird weiter unten bei der

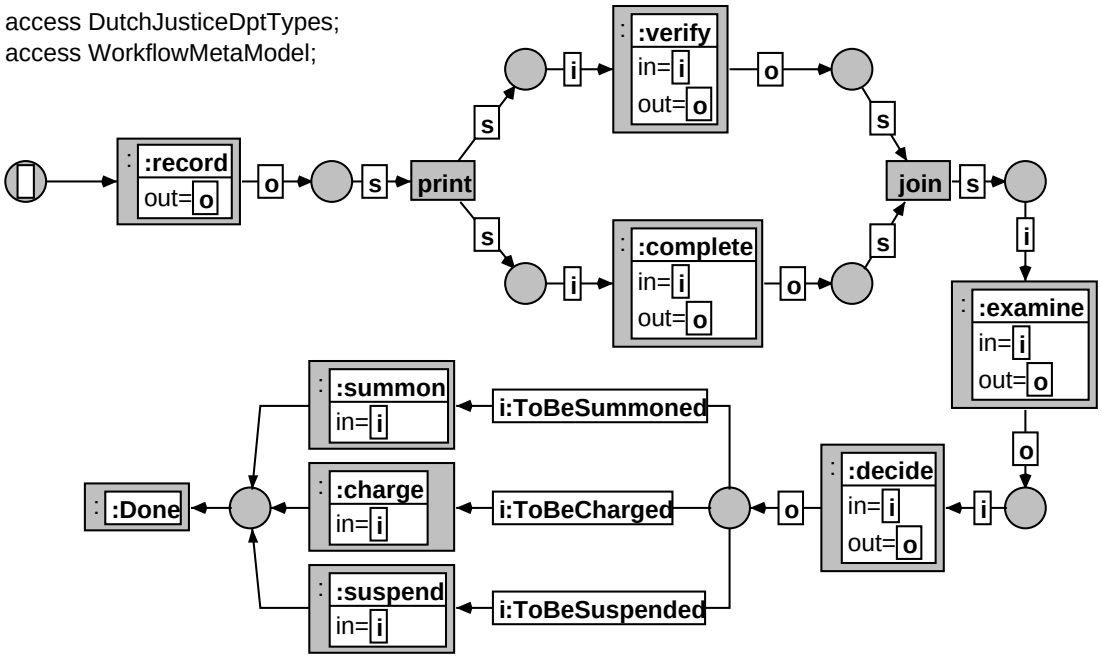

Abbildung 5.3: Der Gerichts-Geschäftsprozeß als HFSNet.

Simulation des Netzes durch ein Umgebungsnetz deutlich. Im folgenden wird die Funktionsweise des Netzes kurz erläutert.

Durch die record-Transition gelangt eine Akte im Initialzustand in das Netz. Die Transition print sorgt für die Erstellung von zwei Ausdrucken oder Kopien der Akte, die durch verify und complete weiterverarbeitet werden. Man beachte, daß die Angabe der speziellen Typen für die Features by, in und out nach den Notationskonventionen aus Abschnitt 2.3.5 nicht nötig sind. Die Transition join unifiziert die durch die nebenläufige Bearbeitung erzeugten Zustände der Akte und führt so den Steuer- und Datenfluß wieder zusammen.

Nach examine findet mit decide ein interessanter Fall statt, in dem eine Entscheidung getroffen wird. In der Simulation (siehe unten) wird von der decide-Transition eine unbestimmte Entscheidung abgelegt und so genaugenommen erst durch die nichtdeterministische Auswahl einer der Folgetransitionen der Entscheidungszustand hergestellt. Wird das Netz aber mit den Renew-Workflow-Erweiterungen eingesetzt, so soll das Ergebnis der Entscheidung des Workflow-Teilnehmers durch den von der Workflow-Umgebung simulierten Startkanal übergeben werden.

Da die Aufgaben-Transitionen im FS-Workflow-Netz als Zielkanäle modelliert werden, kann das Netz ohne entsprechende Startkanäle nicht schalten. Eine Möglichkeit stellt die oben genannte Ausführung als Workflow mit den Renew-Workflow-

Erweiterungen dar. Um eine *Simulation* von Geschäftsprozessen vornehmen zu können, kann die Workflow-Umgebung aber auch von einem weiteren Netz simuliert werden.

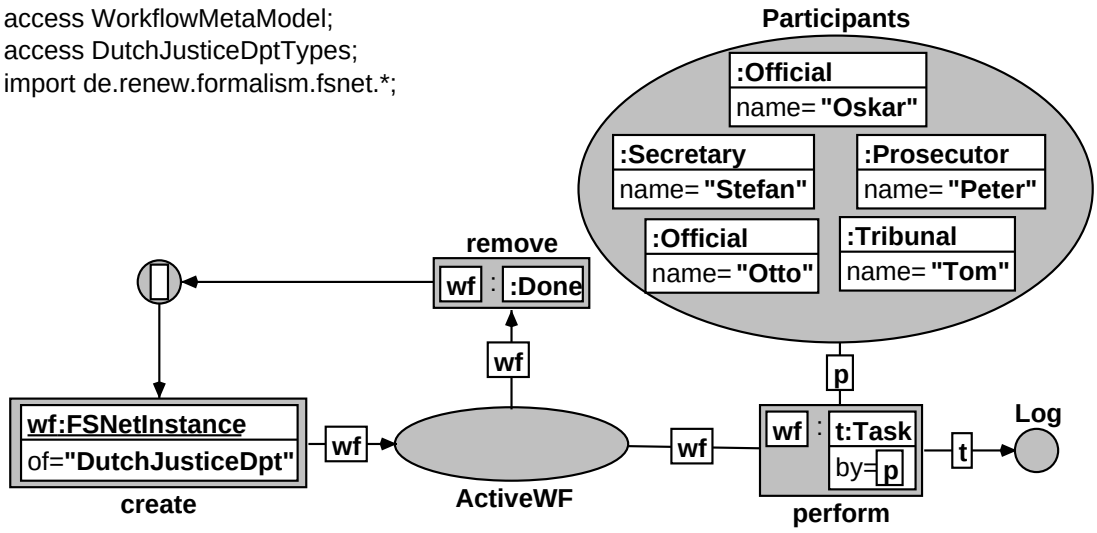

Abbildung 5.4: Ein FSNet zur Simulation einer Workflow-Umgebung für den Geschäftsprozeß aus Abbildung 5.3.

In Abbildung 5.4 ist ein Beispiel für ein FSNet zur Simulation einer Workflow-Umgebung für den Geschäftsprozeß aus Abbildung 5.3 gezeigt. Die Transition create erzeugt ein neues Netzexemplar des FSNet aus Abbildung 5.3 und legt dieses in die Stelle ActiveWF. Jedes einzelne Workflow-Exemplar wird demnach durch genau ein FSNet-Exemplar dargestellt. Dadurch ist es möglich, beliebig viele Workflow-Exemplare gleichzeitig zu behandeln. Durch die voneinander unabhängigen Markierungen von Netzexemplaren kann es nicht wie bei gefärbten Netzen zu unerwünschten Wechselwirkungen zwischen den Workflow-Exemplaren kommen.

Die Transition remove erkennt durch den Startkanal mit dem Parameter [Done] (aus dem Workflow-Meta-Modell), daß eins der aktiven Workflow-Exemplare beendet ist und entfernt dieses. Da nur dann ein neues Workflow-Exemplar erzeugt wird, gibt es in diesem Beispiel einer Workflow-Umgebung nur ein Workflow-Exemplar zur Zeit.

Die Transition perform in Abbildung 5.4 versucht eine Synchronisation mit dem aktiven Workflow-Netzexemplar aus der Stelle ActiveWF. Transition perform kann eine beliebige Aufgabe (Typ Task) auslösen, für die allerdings ein Teilnehmer aus der Stelle Participants zur Verfügung stehen muß, der die von der Aufgabe geforderte Rolle einnehmen kann. Beim Schalten der Transition wird das um die Rolle spezialisierte Task in der Stelle Log abgelegt (Knotenreferenz [t]), um zu protokollieren,

welche Aufgabe von wem ausgeführt wurde. *Wann* die Aufgabe ausgeführt wurde, ist in diesem Beispiel anhand der Markierung der Stelle Log nicht zu erkennen.

Durch die Sematik der Testkante (siehe Abschnitt 3.2.4) ist ein nebenläufiges Schalten der Transition perform möglich, wenn die Aufgaben im Workflow-Exemplar nebenläufig aktiviert sind *und* in der Stelle Participants ausreichend „menschliche Ressourcen“ zur Verfügung stehen.

Der folgende Abschnitt zeigt anhand eines Beispiels den Einsatz von FSNets zur Modellierung von Geschäftsprozessen mit strukturierten Dokumenten.

5.1.2 Online-Bestellungsverarbeitung

Das in diesem Abschnitt mit FSNets modellierte Beispiel eines Geschäftsprozesses zur Online-Bestellungsverarbeitung zeichnet sich durch die Behandlung strukturierter Dokumente aus. Das Beispiel basiert auf einer Fallstudie von [Weitz 1998b], die im Original mit den in Abschnitt 3.4.5 vorgestellten SGML-Netzen durchgeführt wurde. Da SGML-Netze und FSNets gewisse Parallelen zeigen, bietet sich während der Diskussion des Beispiels ein Vergleich beider Ansätze an.

Die Motivation für ein weiteres Beispiel ist, daß im vorherigen Abschnitt nicht die *Struktur* der durch den Workflow verarbeiteten Dokumente, sondern nur deren Zustand modelliert wurde. SGML-Netze werden in [Weitz 1998b] dadurch motiviert, daß sie in der Lage sind, die Bearbeitung strukturierter Dokumente in Workflows zu modellieren. In diesem Abschnitt wird gezeigt, daß dies auch mit FSNets möglich ist.

Das Beispiel aus [Weitz 1998b] wird im folgenden beschrieben. Dabei wird mit einem vereinfachten Modell begonnen, das nach und nach erweitert wird, um zu zeigen, daß sich auch komplexere Modelle mit Hilfe von FSNets darstellen lassen.

Die zu modellierende Online-Bestellungsverarbeitung sieht vor, daß ein *Kunde* eine *Bestellung* über das Internet an einen *Versandhandel* für Computer-Hardware verschicken kann. Im System des Versandhändlers treffen die Bestellungen als Dokumente ein, deren Struktur in [Weitz 1998b] durch eine DTD gegeben ist. Die beim Versandhändler verfügbaren *Produkte* werden ebenfalls durch Dokumente mit einer entsprechenden DTD spezifiziert. Der Versandhändler prüft, ob die bestellten Produkte im *Lager* vorrätig sind. Ist dies der Fall, so wird ein *Lieferschein* ausgestellt und zusammen mit den bestellten Produkten versandt. Die nicht vorrätigen Produkte werden beim *zentralen Warenlager* durch eine *Nachbestellung* angefordert. Auch Lieferschein und Nachbestellung werden über DTDs spezifiziert. Vom zentralen Warenlager gelieferte Produkte werden im Lager abgelegt.

In SGML-Netzen wird die informationsorientierte Modellierung durch SGML-Dokumenttyp-Definitionen (DTDs) und SGML-Dokumente über diesen DTDs vorgenommen (siehe Abschnitt 3.4.5). FSNets decken die wesentlichen Möglichkeiten der Informationsmodellierung von SGML ab, da DTDs auf Typsysteme und Dokumente auf Feature Structures abgebildet werden können. Diese Abbildung soll hier

nicht allgemein betrachtet, sondern an dem gegebenen Beispiel veranschaulicht werden. Grob gesagt werden Sequenzen durch Listen, Alternativen durch Subtypen und andere SGML-Konstrukte durch Attribute ersetzt.

Die eigentliche Herausforderung besteht darin, die SGML-Muster in entsprechende Transitionsregeln eines FSNets zu übersetzen. Das Prinzip des Musterabgleichs bei SGML-Netzen weist bereits eine gewisse Ähnlichkeit zum Anwenden einer Transitionsregel bei FSNets auf. Die Verwendung von Knotenreferenzen ähnelt der Verwendung von Variablen in SGML-Netzen. Da SGML-Netze selbst nicht formal definiert sind, soll auch die Abbildung von SGML-Mustern auf FSNet-Transitionsregeln hier nur anhand des Beispiels erfolgen.

Die Online-Bestellungsverarbeitung wird dazu in mehreren Schritten mit einem FSNet modelliert. Als Vorgehen wird erst eine Informationsmodellierung, anschließend eine Prozeßmodellierung vorgenommen, ähnlich wie im Originalbeispiel, in dem zuerst die Dokumentenstrukturen beschrieben werden und erst dann der Workflow modelliert wird.

Bestellungen mit einzelnen Produkte

Im ersten Schritt modellieren wir einen einfachen Bestellvorgang, in dem pro Bestellung nur ein Produkte erlaubt ist.

Abbildung 5.5 zeigt das Informationsmodell zu diesem ersten Schritt als Konzeptsystem. Eine Bestellung (Order) für ein Produkt (Product) wird von einer Person (Person) in der Rolle als Kunde (Attribut customer) spezifiziert. Alle Produkte haben einen Namen (name), eine Produktnummer (serialno), eine Marke (brand) und einen Preis (price), wobei diese durch Java-Basistypen bzw. Java-`String`s dargestellt werden. Spezielle Produktgruppen, im Beispiel durch Mousepad, Cable, CPU und Mainboard angedeutet, können über den Vererbungsmechanismus der Typmodellierung weitere Attribute hinzufügen, beispielsweise die Länge (length) des Kabels etc. Die Typen Lieferschein (DeliveryNote) und Nachbestellungsanfrage (Request) werden hier als Spezialisierung bzw. Subkonzepte von Order vereinbart, da sie stets aus einer Bestellung entstehen. Diesen Sachverhalt könnte man ebensogut durch eine Assoziation von DeliveryNote und Request zu Order modellieren. Request fügt ein weiteres Attibut hinzu, das als Absender des lokalen Versandhändlers dient, wenn er Bestellungen an das zentrale Warenlager weiterleitet.

Abbildung 5.6 stellt die vereinfachte Online-Bestellungsverarbeitung als FSNet basierend auf dem oben beschriebenen Konzeptsystem dar. Dieses Netz sieht in der Struktur dem in [Weitz 1998b] entwickelten SGML-Netz sehr ähnlich, weswegen im folgenden vor allem die Unterschiede zwischen den beiden Modellierungsarten und die Umsetzung der SGML-Muster in Transitionsregeln diskutiert werden.

Jedem Dokumenttyp aus dem SGML-Netz entspricht ein Konzept aus dem Konzeptsystem; die Stellen tragen die entsprechende Typinformation (in Form von Feature Structures). Die Kanten sind statt mit SGML-Mustern mit Feature Structures

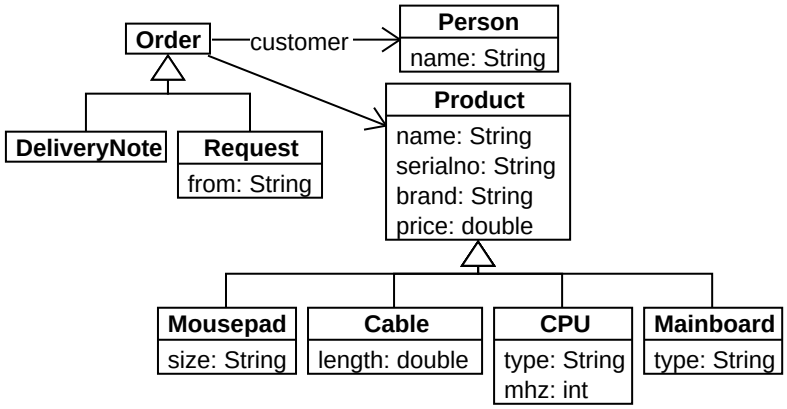

Abbildung 5.5: Das Konzeptsystem für die vereinfachte Online-Bestellungsverarbeitung.

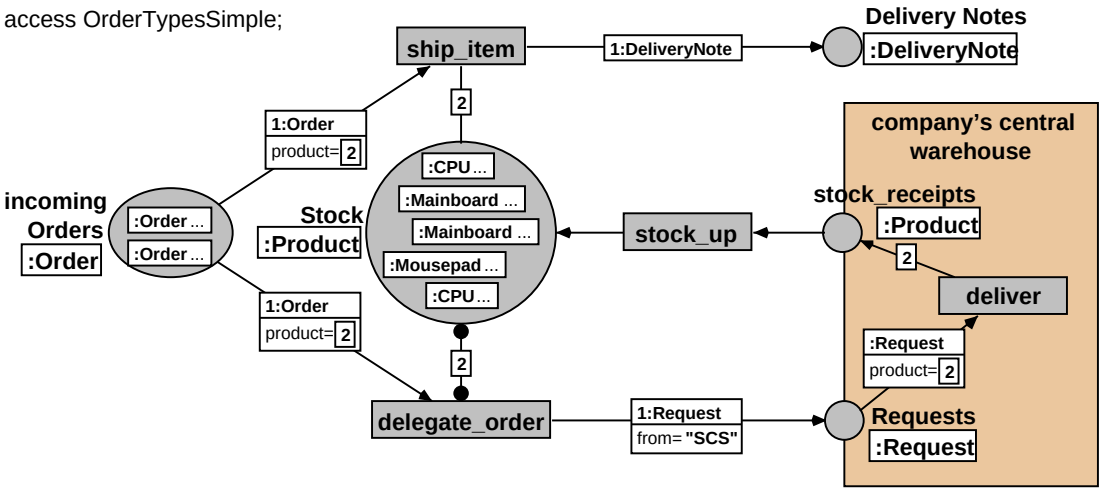

Abbildung 5.6: Die vereinfachte Online-Bestellungsverarbeitung als FSNet modelliert.

beschriftet, die in der FSNet-Modellierung den Zweck des Musterabgleichs erfüllen.

Die drei Transitionen erfüllen ihre Aufgaben durch die im folgenden erläuterten Kantenanschriften.

Der Kantenausdruck an der Eingangskante von incoming Orders zu ship_item führt dazu, daß Knotenreferenz 2 auf die Unterstruktur der Eingangsmarke zeigt, die das Produkt der Bestellung enthält. Durch die Eingangskante von Stock wird beim Schalten der Transition eine Marke für ein Produkt aus der Lager-Stelle entfernt, deren Daten mit denen aus der Produkt-Unterstruktur der Anfrage unifizierbar sind. Als Ausgangsmarke wird ein Lieferschein erzeugt, welcher der Struktur der Eingangs-

marke entspricht, die mit den Produktdaten unifiziert wurde und zum Typ DeliveryNote verfeinert wurde. Damit wird ein Lieferschein erstellt, der die Lieferung genau beschreibt.

Die Transition delegate_order ist wie in der Modellierung als SGML-Netz für den Fall zuständig, in dem kein auf die Beschreibung aus der Bestellung passendes Produkt auf Lager ist. Ebenso wie bei SGML-Netzen wird diese Negativbedingung durch eine Inhibitorkante geprüft, was eine sehr prägnante Schreibweise erlaubt. Die Anschriften der Eingangskanten gleichen denen der Transition ship_item, nur das Ergebnis ist ein anderes: In diesem Fall soll eine Nachbestellungsanfrage (Request) an das zentrale Warenlager geschickt werden. Dafür wird der Typ der Bestellung von Order auf Request verfeinert und das in diesem Typ zusätzlich erlaubte Attribut für den Absender gesetzt.

Die Transition stock_up schließlich ist sehr einfach strukturiert. Sie befördert in der Stelle stock_receipt ankommende nachbestellte Produkte in das Lager. Man beachte dazu daß unbeschriftete Kanten aus die Wurzel der Transitionsregel verweisen (siehe Abschnitt 4.1.2).

Ein wichtiger Unterschied zu den SGML-Netzen ist, daß dort streng zwischen Markenstruktur (Dokumente) und Kantenanschrift (SGML-Muster) unterschieden wird. In FSNets werden sowohl Dokumente als auch Muster durch Feature Structures beschrieben. Dies ist einerseits ein allgemeinerer und einheitlicherer Ansatz, andererseits wird vom Modellierer mehr Disziplin verlangt. Als Beispiel dafür besteht in dem oben präsentierten FSNet die Gefahr, die Marken der Anfangsmarkierung in der Stelle Stock *allgemeiner* zu wählen als die in der Stelle Orders, was nicht im Sinne der Modellierung wäre. Die Bestellungen kann man sich wie Anfragen an eine Datenbank vorstellen, während die Produkte die Daten selbst darstellen. Wären die Daten allgemeiner als die Anfrage, würde in diesem Modell bei der Unifikation die speziellere Information geliefert, obwohl diese gar nicht Teil der Produktbeschreibung im Lager ist. Um dieses Problem zu beheben, müßte man Wächter an die Transitionen setzen, die explizit auf Subsumption, nicht nur wie durch den Formalismus vorgegeben auf Unifizierbarkeit prüfen.

Beim Vergleich mit Datenbanken fällt ein weiterer Unterschied auf, den man sich bei der Modellierung mit FSNets (bzw. allgemein mit Petrinetzen) klar machen muß: Wenn es mehrere passende Marken gibt, so wird *eine davon* nicht-deterministisch gewählt. Datenbankabfragesprachen wie SQL liefern üblicherweise die gesamte Menge der passenden Antworten. Dieses Verhalten kann man natürlich auch mit FSNets nachbilden, dann wird die Lösung allerdings etwas aufwendiger, da nicht mehr der „eingebaute“ Auswahlmechanismus aus einer Multimenge von Marken verwendet werden kann.

Die nicht-deterministische Auswahl hat weiterhin zur Folge, daß nicht, wie man annehmen könnte, die speziellste der passenden Marken beim Schalten gewählt wird. Ein weiterer Modellierungsfehler wäre es im Beispiel also, wenn sich verschiedene Produktbeschreibungen subsumieren, da in diesem Fall immer dann, wenn die spe-

ziellere Beschreibung paßt, auch die allgemeinere geliefert werden könnte.

Bestellungen mit mehreren Produkten und Nachbestellung

Im zweiten Schritt soll die oben angegebene Problemstellung aus [Weitz 1998b] in ihrer vollen Komplexität übernommen und sogar noch erweitert werden. Im einzelnen sollen nun Listen von Produkten in einer Bestellung enthalten sein können. Außerdem soll bei der Delegation an das zentrale Warenlager die Ausführung der Bestellung vom Versandhändler selbst durchgeführt werden.

In der Informationsmodellierung ergibt sich daraus die Änderung, daß die Assoziation zwischen Bestellung und Produkt eine mehrfache Multiplizität erhält. In der Typmodellierung für Feature Structures werden Mehrfachassoziationen auf parametrische Listentypen abgebildet (siehe Abschnitte 2.1.1 und 2.3.5). Bestellungen werden in die disjunkten Subkonzepte ImmediateOrder und PendingOrder unterschieden. Damit wird gekennzeichnet, ob zu einer Bestellung bereits eine Nachbestellung an das zentrale Warenlager versandt wurde. Noch sauberer wäre hier eine separate Modellierung des Zustands der Bestellung, die sich aber für ein so simples Zustandsmodell nicht lohnt.

Nachbestellungen des Versandhändlers sollen weiterhin auf einzelne Produkte beschränkt bleiben. Da im geänderten Szenario der Versandhändler selbst die Auslieferung der nachbestellten Produkten übernimmt, ist die Assoziation von Request zu Product überflüssig. Request wird deshalb nicht mehr als Subtyp von Order modelliert und erhält die zuvor geerbte Assoziation zu Product. Abbildung 5.7 zeigt das resultierende Konzeptsystem.

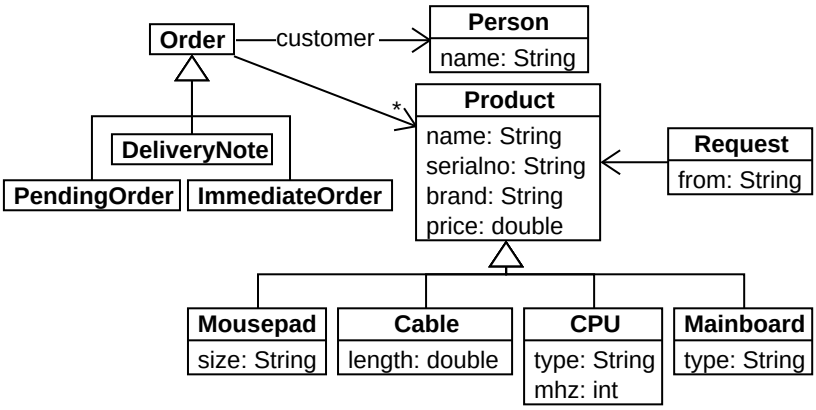

Abbildung 5.7: Das Konzeptsystem für die Online-Verarbeitung von Bestellungen mit mehrerer Produkten.

Abbildung 5.8 stellt die Online-Verarbeitung von Bestellungen mit mehrerer Produkten als FSNet dar. Für jede Bestellung wird von der Transition new_note ein

zunächst leerer Lieferschein angelegt. Jede Bestellung kann von der ship_product-Transition verarbeitet oder durch delegate_order versandt werden, eine PendingOrder aber nicht, damit ein gewünschtes Produkt nicht mehrfach nachbestellt wird.

Objektorientierte Netzmodellierung der Online-Bestellungsverarbeitung

In Abschnitt 3.3.3 wurden Referenznetze als eine Möglichkeit der objektorientierten Strukturierung von prozeßorientierten Modellen vorgestellt. Andere höhere Petrinetze bieten dagegen meist zur Strukturierung komplexer Modelle nur hierarchische Verfeinerung. Während man in der Softwareentwicklung schon längst von der funktionalen Dekomposition zur Objektorientierung übergegangen ist ([Coad und Yourdon 1991, Moldt 1996]), steht dieser Schritt in der Prozeßmodellierung immer noch aus. Referenznetze können uns diesem Ziel zumindest einen Schritt näher bringen.

Wie in Abschnitt 5.1 sollen deshalb auch dieses Beispiel die in Abschnitt 4.4 eingeführten HFSNets eingesetzt werden. Während bei der Workflow-Ausführung und -Simulation das Dynamische Anlegen von Netzexemplaren im Vordergrund stand, geht es in der alternativen Modellierung dieses Beispiels um die Zerlegung eines Modells in einzelne Netzobjekte. Dies zeigt die aus der Objektorientierung bekannten Vorteile der Strukturierung und der Wiederverwendung ([Gamma et al. 1995] und Teil 4 in [Ciancarini 1999]). Der Online-Bestellvorgang soll nun so modelliert werden, daß die Verwaltung des Lagers in ein eigenes Objekt ausgelagert wird. Zu den Aufgaben des Lagers gehört auch die Verwaltung der Information, welche Produkte überhaupt existieren bzw. nachbestellbar sind.

Dafür wird das FSNet aus Abbildung 5.8 in ein HFSNet-Netzsystem überführt, das aus zwei Netzen OrderProcessing und Stock besteht, wobei OrderProcessing als Hauptnetz fungiert und Stock von diesem instantiiert und aufgerufen wird.

Für jeden der zur Kommunikation genutzten synchronen Kanäle wird ein neuer Typ definiert. Die Schnittstelle zwischen dem Hauptnetz und dem Lager besteht aus mehreren Kanälen, die sich alle auf genau ein Produkt beziehen. Abbildung 5.9 zeigt die Erweiterung zum Konzeptsystem aus Abbildung 5.7. Für die Kanäle wird zunächst ein abstrakter Typ LinkWithProduct eingeführt, der ein Attribut prod vom Typ Product (aus dem anderen Konzeptsystem) aufweist. Die drei möglichen Kanalnamen sind

- Get für das Abrufen eines vorhandenen Produkts,
- Unavailable für die Abfrage, ob ein Produkt zwar prinzipiell lieferbar, aber nicht im lokalen Lager vorhanden ist,
- StockUp für das Auffüllen des Lagers mit nachgelieferten Produkten.

Es werden nun die Änderungen am Netz OrderProcessing, zu sehen in Abbildung 5.10, beschrieben, danach das neue Netz Stock näher betrachtet. Die Stelle

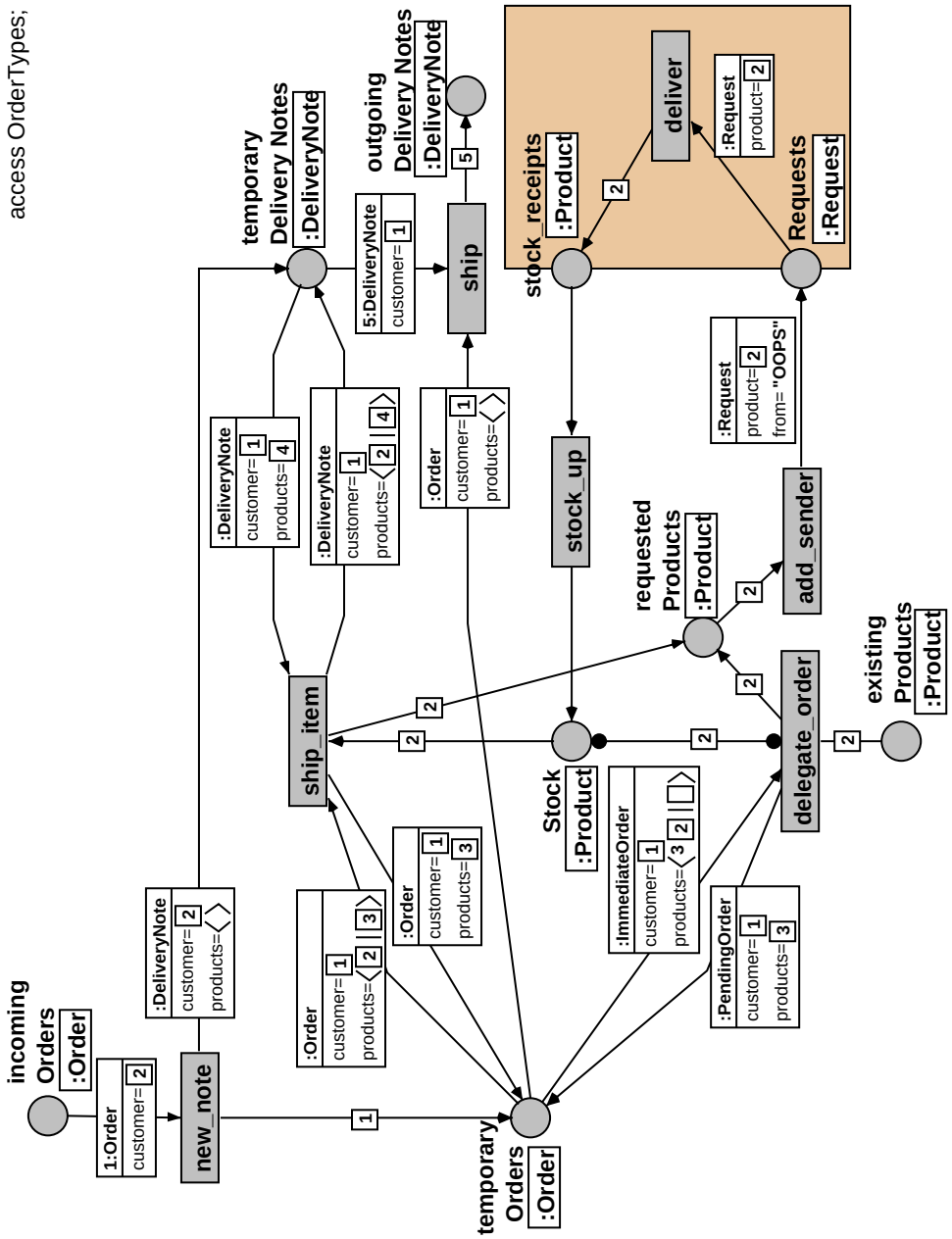

Abbildung 5.8: Die Online-Verarbeitung von Bestellungen mit mehrerer Produkten als FSNet modelliert.

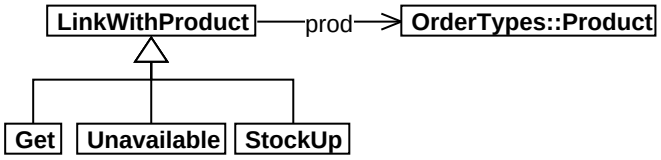

Abbildung 5.9: Die Erweiterung des Konzeptsystems der Online-Bestellungsverarbeitung aus Abbildung 5.7 um Konzepte für synchrone Kanäle.

Stock enthält als Initialisierungsanschrift eine Java-Feature-Structure, die zu einer Referenz auf ein neues Netzexemplar des Lagers (Netz Stock) ausgewertet wird. Die Transitionen ship_item, stock_up und delegate_order, die zuvor direkt auf die Produkte im Lager zugegriffen haben, delegieren die entsprechenden Befehle nun über synchrone Kanäle an das Stock-Netzexemplar. Die Netzreferenz wird über Testkanten zugegriffen, da diese selbst nicht verändert wird.

Während beim synchronen Kanälen : [StockUp] kein Unterschied zu dem Verhalten in Referenznetzen erkennbar ist, zeigen die Kanäle : [Get] und : [Unavailable] das spezielle Verhalten in HFSNets. Hier wird die Unifikation ausgenutzt, indem Information in beide Richtungen fließt: Die in der Bestellung enthaltene Produktanfrage wird an das andere Netzexemplar übermittelt, welches daraufhin ein komplett beschriebenes Produkt liefert, falls ein solches vorrätig ist. Dieser Informationsfluß in beide Richtungen ist mit einem Referenznetz zwar auch möglich, aber nur mit zwei verschiedenen Variablen, von denen jeweils eine in einem der Netze belegt wird. Eine Feature Structure ersetzt damit die gesamte Liste von Parametern.

Das Netz Stock ist insofern eine objektorientierte Modellierung eines Lagers, als es zumindest einige der Anforderungen an ein Objekt erfüllt: Es kapselt Daten (Marken in Stellen), die nur über Operationen (synchrone Kanäle) zugreifbar sind. Es kann mehrfach und dynamisch (zur Laufzeit) instantiiert werden und jedes Exemplar besitzt seine eigene Identität, über die es referenziert werden kann, und seinen eigenen Zustand (Markierung des Netzexemplars). Es ist in verschiedenen Systemen wiederverwendbar.

Inhaltlich bietet das in Abbildung 5.11 gezeigte Netz Stock folgende Schnittstelle. Über den Zielkanal : [Get] können andere Objekte ein Produkt abfragen. Das Produkt wird zugleich aus dem Lager entfernt. : [Stockup] kann von einem Zulieferer aufgerufen werden, der das Lager auffüllen möchte. [Unavailable] schließlich fragt ab, ob es zu einer Abfrage ein Produkt gibt, das vom zentralen Warenlager nachlieferbar, aber momentan nicht im Lager vorhanden ist. Ein naheliegendes Gegenstück, das in der vorliegenden Lösung nicht berücksichtigt wurde, wäre ein Zielkanal : [Available], der die Beschreibung eines vorhandenen, auf die Abfrage passenden Produkts liefert, ohne dieses aus dem Lager zu entfernen.

Der Vorteil an der Kapselung ist, daß die Funktionalität auf verschiedene Art

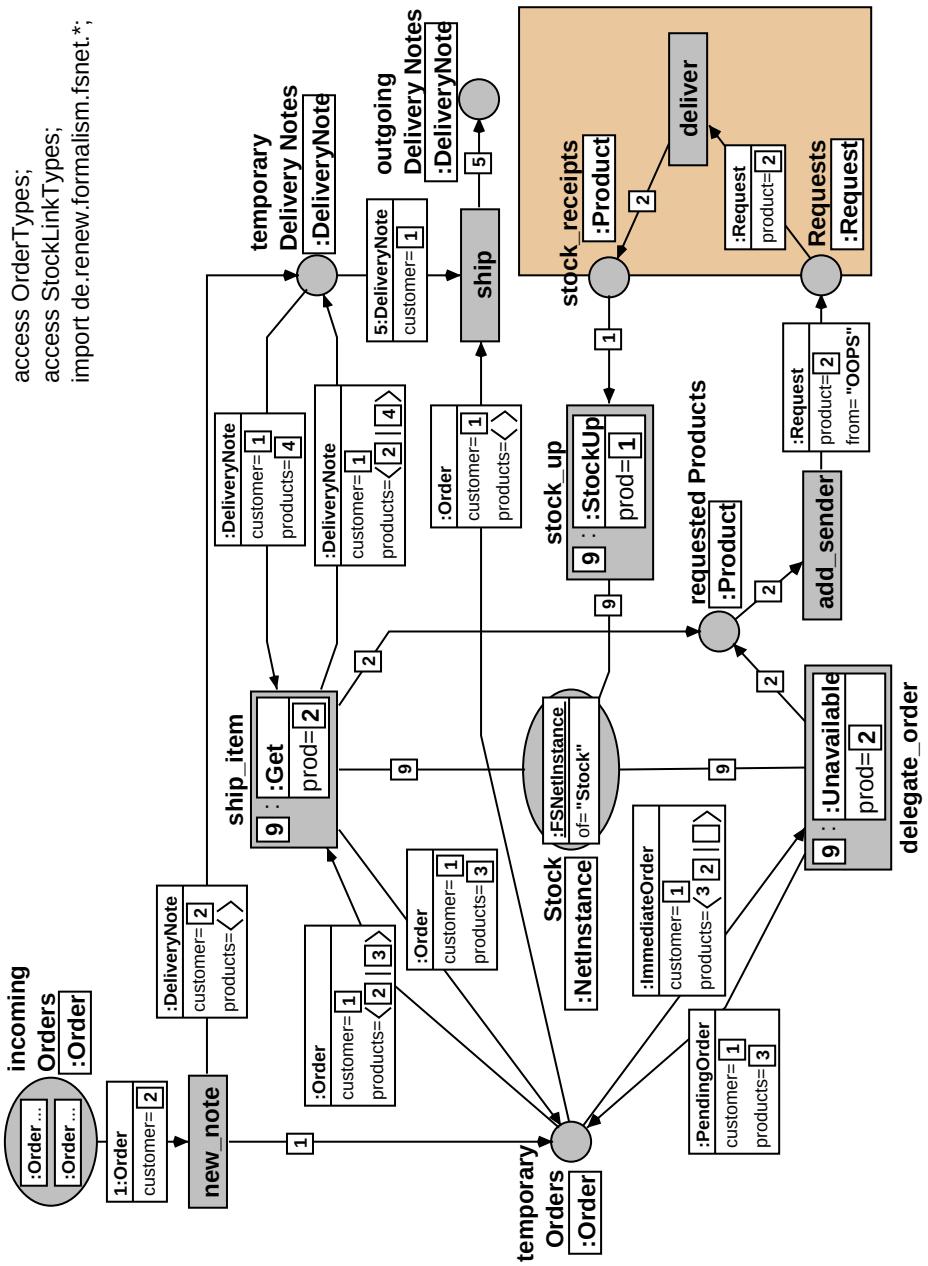

Abbildung 5.10: Die Online-Bestellungsverarbeitung aus Abbildung 5.8 als HFSNet. Das Lager (Stock) wird als eigenes Netzobjekt modelliert.

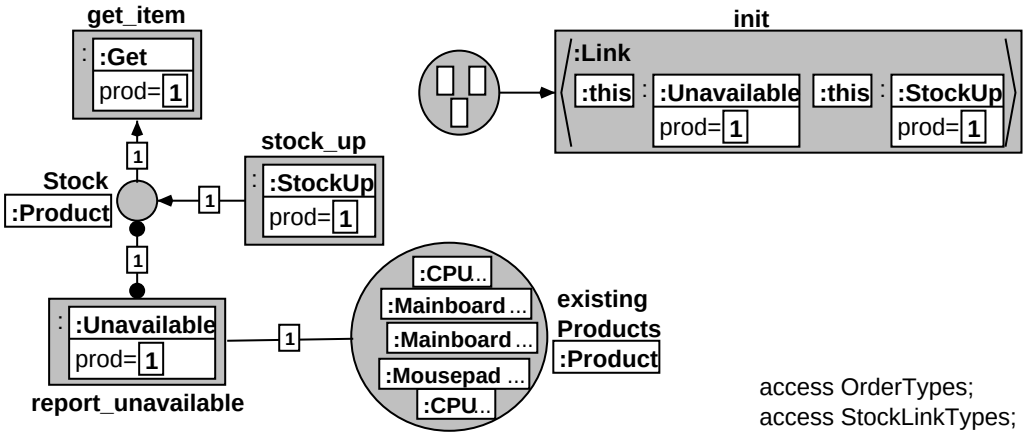

Abbildung 5.11: Das Lager zur Online-Bestellungsverarbeitung aus Abbildung 5.10 als eigenes Netzobjekt modelliert.

und Weise erbracht werden kann. Es werden keine Annahmen über die Repräsentation des Lagers gemacht, sondern nur darüber, mit welchen Operationen darauf zugegriffen werden kann. In der hier gezeigten Lösung wird jedes Produkt durch eine eigene Marke dargestellt. Eine Alternative, welche dieselbe Schnittstelle anbieten könnte, wäre die Repräsentation des gesamten Bestands als eine Marke, die aus einer Liste von Produkten besteht. Die Operationen hätten dann die Aufgabe, diese Liste zu durchsuchen. Diese Modellierung sollte gewählt werden, wenn man die aus Petrinetz-Sicht unschöne Inhibitorkante vermeiden möchte (siehe Abschnitt 3.2.4).

In diesem Abschnitt wurde ein Beispiel, das in [Weitz 1998b] mit SGML-Netzen modelliert wurde, auf FSNets übertragen. In Abschnitt 3.4.5 wurde an SGML-Netzen vor allem die mangelnde Formalisierung kritisiert, die FSNets nicht vorgeworfen werden kann. FSNets besitzen eine formal definierte Semantik, sowohl für die Aspekte der Datenmodellierung (Feature Structures) als auch für die Schaltregel.

Mit SGML halten sich SGML-Netze an eine spezielle Technik, die standardisiert ist. FSNets basieren auf Feature Structures, deren formale Semantik in dieser Arbeit wiedergegeben und um zusätzliche Operatoren erweitert wurde. Standardisiert sind Feature Structures selbst nicht. In dieser Arbeit wird aber eine Abbildung von Feature Structures auf Konzepte und Notationen des Modellierungsstandards UML gezeigt.

Während SGML auf die Darstellung von Dokumenten spezialisiert ist, handelt es sich bei Feature Structures um eine allgemeine Technik zur Repräsentation von Wissen oder Objektsichten. Durch Konzepte wie Vererbung und Assoziationen gehen die Möglichkeiten von Feature Structures über die von SGML hinaus, das lediglich streng hierarchische Strukturen und basiswertige Attibute unterstützt.

Die Idee des Musterabgleichs zwischen Kantenausdrücken und Marken in SGML-Netzen wurde in Abschnitt 3.4.5 grundsätzlich positiv beurteilt. Die Übertragung des Beispiels auf FSNets zeigt, wie auch komplexe Musterabgleiche durch Feature Structures dargestellt werden können. Dadurch, daß Marken und Kanten- sowie Transitionsanschriften durch Feature Structures beschrieben werden, entstehen neue Möglichkeiten gegenüber SGML-Netzen. Beispielsweise muß die Bestellung eines Produkts nicht anhand eines eindeutigen Schlüssels erfolgen, sondern kann wiederum aus einem Muster bestehen, das mit den vorrätigen Produkten verglichen wird. In SGML-Netzen sind Muster nur an Kanten erlaubt und damit nur statisch spezifizierbar.

Weiterhin wurde in Abschnitt 3.4.5 kritisiert, daß SGML-Netze das grundlegende Prinzip der Markenkonsumierung in Petrinetzen nicht einhalten. Der Nachteil ist, daß das Schaltverhalten des SGML-Netzes nicht ein spezialisiertes Verhalten des S/T-Netzes ist, das man erhält, wenn alle Anschriften weggelassen werden. FSNets sind so definiert, daß jede ihrer Schaltfolgen auch eine Schaltfolge des unterliegenden S/T-Netzes ist (aber nicht umgekehrt). Damit lassen sich bedingt Analyseverfahren von S/T-Netzen für FSNets einsetzen. Gibt es beispielsweise im unterliegenden S/T-Netz nur endliche Schaltfolgen, so muß dies auch für das FSNet der Fall sein. Auch Stelleninvarianten lassen sich, was die *Anzahl* der Marken betrifft, auf FSNets übertragen.

5.2 E-Commerce: elektronische Verträge

Electronic Commerce oder kurz *E-Commerce* umfaßt den gesamten Bereich wirtschaftlicher oder kommerzieller Nutzung von Computernetzen wie dem Internet. Dazu zählt sowohl der Handel zwischen Unternehmen und ihren Kunden (*business to consumer*, B2C) als auch direkt zwischen Unternehmen (*business to business*, B2B).

In [Merz et al. 1999b] wird der Begriff des *elektronischen Marktes* verwendet, der im wesentlichen ein Synonym für E-Commerce darstellt. Durch die Eingrenzung auf elektronische *Dienste*märkte (EDM, *service markets*) wird betont, daß durch elektronische Märkte vor allem Dienste angeboten und genutzt werden. Merz beschreibt ein Dienstemodell, das eine formalisierte Spezifikation von Diensten in elektronischen Märkten und damit eine verbesserte Klassifikation und Vergleichbarkeit von Dienstanbietern erlaubt. Dienstanbieter und -nachfrager schließen demnach einen *elektronischen Vertrag* ab, der den zu erbringenden Dienst genau spezifiziert. Im EU-Projekt COSMOS ([COSMOS 1999]) wurde eine Software-Architektur für ein EDM-Modell konzeptualisiert und prototypisch implementiert.

Im folgenden Abschnitt wird über die Konzepte und die Architektur von COSMOS als ein Beispiel für die Realisierung eines EDM-Modells ein grober Überblick gegeben. Auf den Bereich, in dem die in dieser Arbeit vorgestellten Modellierungs-

techniken und Werkzeuge eingesetzt wurden, wird im darauf folgenden Abschnitt genauer eingegangen.

5.2.1 COSMOS: Eine Infrastruktur für elektronische Dienstemärkte

COSMOS (*Common Open Service Markets fOr Small and medium enterprises*, siehe [COSMOS 1999]) wurde von Mai 1998 bis Mai 2000 als EU-Projekt von Projektpartnern aus Industrie, Forschung und dem Anwendungsgebiet durchgeführt. Das Ziel von COSMOS war, die zentralen Konzepte und systemtechnischen Anforderungen an eine Infrastruktur für elektronische Dienstemärkte herauszuarbeiten und in einer prototypischen Software-Plattform umzusetzen. Dabei wurde die Konzeption der Architektur und ein Teil der Implementierung in der Arbeitsgruppe *verteilte Systeme* am Fachbereich Informatik der Universität Hamburg geleistet.

Als wesentliche Ergebnisse der Konzeption entstanden eine genauere Spezifikation elektronischer Dienste und Verträge, der Entwurf einer Softwarearchitektur, ein objektorientiertes Modell elektronischer Verträge und ein erweiterter Ansatz zum Makeln (*brokerage, trading*) in elektronischen Dienstemärkten. Diese vier Aspekte werden im folgenden in Anlehnung an [Griffel et al. 1998] näher betrachtet. Darauf folgt eine detailliertere Darstellung der Ausführung elektronischer Verträge.

Elektronische Märkte und Verträge

Jede kommerzielle Transaktion in einem EDM wird von einem elektronischen Vertrag begleitet, der den Dienst und die Rechte und Pflichten der teilnehmenden Parteien genau beschreibt. Zu den Pflichten zählt beispielsweise, eine Zahlung zu leisten, zu den Rechten, dafür die Lieferung eines Guts zu erhalten.

COSMOS verfolgt bei der Abwicklung einer Geschäftstransaktion einen aus der Wirtschaftswissenschaft bekannten Ansatz ([Schmid 1993]), der in Abbildung 5.12 veranschaulicht ist.

- In der *Informationsphase* (*information phase*) suchen Dienstanbieter und -nutzer auf dem elektronischen Markt nach entsprechenden Angeboten oder Nachfragen, die nach Produktspezifikation, Preisen und weiteren Kriterien bewertet werden.

- Haben sich verschiedene Marktteilnehmer zusammengefunden, werden in der *Verhandlungsphase* (*negotiations phase*) Angebote und Gegenangebote ausgetauscht. Der Verhandlungsprozeß führt entweder zu einem Zustand, in dem sich die Teilnehmer über die Einzelheiten des Vertrags einig geworden sind, oder aber die Verhandlung wird abgebrochen.

- Nachdem sich alle Verhandlungsteilnehmer durch ihre (elektronische) Unterschrift zur Teilnahme am Vertrag verpflichtet haben, wird in die *Ausführungs-*

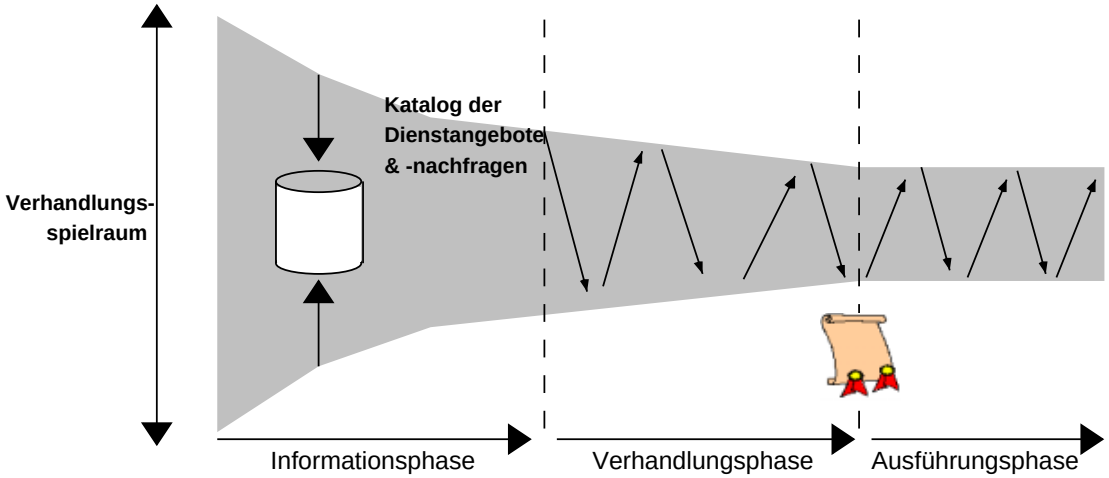

Abbildung 5.12: Die drei Phasen einer Geschäftstransaktion in COSMOS.

phase (*execution phase*, man spricht auch von *contract performance*) übergegangen, während der die Teilnehmer die durch den Vertrag bestimmten Aufgaben zu erfüllen haben.

Da ein Vertrag *ausführungsorientiert* spezifiziert wird, legt er fest, *was wann* von *wem* getan werden muß. Hier ist die enge Verwandschaft zu Geschäftsprozeß- und Workflow-Definitionen erkennbar, wie sie in Abschnitt 3.4 vorgestellt und im Beispiel in Abschnitt 5.1 bereits mit FSNets modelliert wurden. Die Vor- und Nachteile elektronischer Verträge werden in [Griffel et al. 1998] ausführlich diskutiert. Dort werden Verträge weiter klassifiziert und beschrieben durch Aspekte wie die Anzahl der Rollen, die Anzahl der Vertragsparteien, die Anzahl der transaktional durchzuführenden Verträge, Existenz von Querbeziehungen zwischen Verträgen und Automatisierbarkeit der Vertragsteilnahme durch Software-Agenten.

Systemtechnische Realisierung der COSMOS-Infrastruktur

Als technische Basis elektronischer Verträge, an denen verschiedenste Unternehmen und Kunden teilnehmen, dienen sogenannte *Geschäftsobjekte* (*business objects*), für deren Realisierung im Fall von COSMOS die *CORBA business objects architecture* (BOA, [BODTF 1997]) gewählt wurde.

Für die Darstellung von *Katalogen* mit Dienstangeboten und -nachfragen wird eine sehr flexible und dynamische Repräsentation der Daten benötigt. Ein Ansatz allein auf der Basis der BOA erwies sich damit als unzureichend.

In der E-Commerce-Praxis werden Daten dynamischen Typs meist durch untypisierte Name-Wert-Paare (*name-value-pairs*, *property lists*) repräsentiert

([Merz 1999a]). Eine solche Repräsentation ist jedoch aus mehreren Gründen unbefriedigend:

- Ohne eine Typisierung der Daten kann kaum eine angemessene Repräsentation erzielt werden, da die Anwendungsdomäne vollkommen unstrukturiert ist. Eine Vergleichbarkeit verschiedener Angebote ist damit nicht gegeben.
- Untypisierte Name-Wert-Paare erlauben meist nur Werte von Basistypen wie Zeichenketten, Zahlen und Binärobjekte. Komplexere, strukturierte Werte müssen entweder in mehrere Name-Wert-Paare aufgeteilt werden, was große, aber flache und damit unübersichtliche Strukturen erzeugt. Alternativ können wie in Datenbanken oder in XML Schlüsselwerte als Verweise auf andere Objekte interpretiert werden. Auch dies stellt eine Notlösung dar, da keinerlei referenzielle Integrität ([Date 1995]) zugesichert werden kann.
- Untypisierte Datenstrukturen erlauben zwar höhere Flexibilität, aber auf Kosten der Korrektheit. Aus dem Grund, aus dem auch die meisten Programmiersprachen typisiert sind, sollten auch Datenstrukturen typisiert werden: Fehler, die in der untypisierten Variante erst während der Ausführung auftreten, werden in vielen Fällen schon während der Erstellung erkannt.

Zur Erfüllung dieser scheinbar widersprüchlichen Anforderungen nach Flexibilität und Typisierung werden sogenannte Typmanager eingesetzt, die eine dynamische Typverwaltung zur Laufzeit bereitstellen ([Merz 1999b]). In der COSMOS-Architektur ist vorgesehen, ein solches Typmanagement in Form von Feature Structure zu integrieren. Die in Abschnitt 2.4 vorgestellte Feature-Structure-Implementierung, die unter dieser Sicht einen Typmanager darstellt, kam in COSMOS selbst allerdings nicht mehr zum Einsatz, wurde aber in [Häming 1999] als Typmanager in der komponentenbasierten Softwaregenerierung eingesetzt.

COSMOS sieht vor, daß Angebote in einem Katalog in der Form von *Vertragsschablonen* abgelegt werden. In der Informationsphase kann ein Interessent über ein *Suchmuster* (*query*) solche Vertragsschablonen finden, die seinen Vorgaben entsprechen. Dafür unterstützt die COSMOS-Plattform die Teilnehmer des EDM mit einer Makler-Komponente (*broker* oder auch *trader*, siehe dazu [Merz 1999b]). In der Verhandlungsphase wird die durch das Angebot vorgegebene Vertragsschablone als Ausgangspunkt verwendet und sukzessive ergänzt oder modifiziert.

Ein Vorteil der Verwendung von Feature Structure ist, daß nicht nur die Angebote auf der Basis eines dynamischen Typsystems spezifiziert werden können, sondern auch Suchmuster. Der Makler kann im einfachsten Fall mit Hilfe der Subsumptionsfunktion implementiert werden (siehe Abschnitt 2.3.3 und Abschnitt 2.4.3), was aber für realistische Anwendungsfälle nicht ausreichen wird. Von Makler-Mechanismen wird verlangt, daß diese auch Ähnlichkeiten von Suchmuster und Daten bewerten können und so auch im Falle einer nicht vollständigen Übereinstimmung ein Ergebnis liefern. Außerdem können durch eine solche Bewertung mehrere passende

Daten nach ihrer Ähnlichkeit sortiert als Ergebnis geliefert werden. Im Prinzip sind entsprechende Erweiterungen für das in dieser Arbeit vorgestellte Typ- und Feature-Structure-System denkbar, wurden aber nicht weiter verfolgt.

Die Architektur von Cosmos

Um eine flexible und anpaßbare Architektur für elektronische Dienstemärkte zur Verfügung zu stellen, die aus einer größeren Anzahl technisch komplexer Einzelkomponenten besteht, wurde in Cosmos ein modularer Ansatz gewählt.

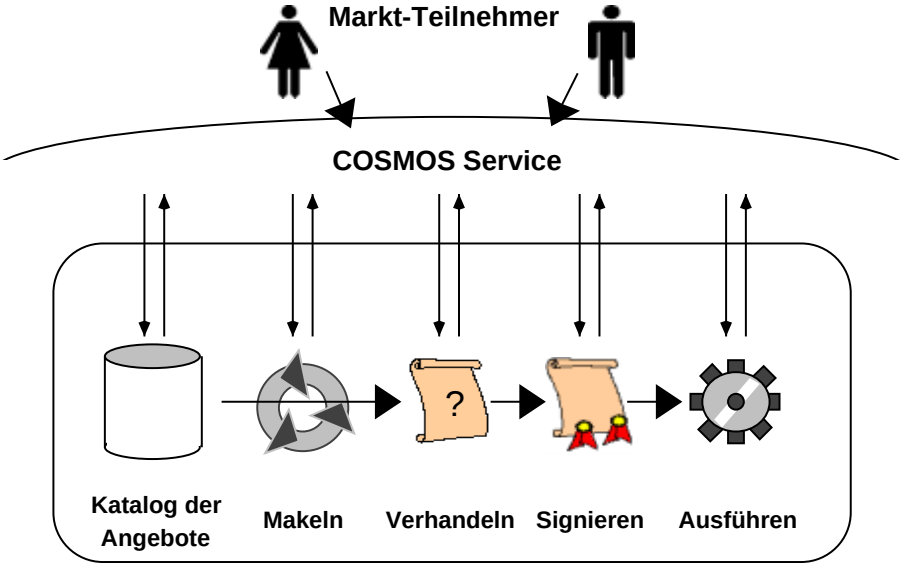

Abbildung 5.13: Komponenten von Cosmos.

Abbildung 5.13 zeigt die wesentlichen Komponenten der Cosmos-Architektur. Es ist kein Zufall, daß diese die oben vorgestellten Geschäfts-Transaktionsphasen widerspiegeln.

- Ein *Angebotskatalog* (*offer catalogue*) enthält wie oben dargestellt verschiedene Dienstangebote und -nachfragen in Form von Vertragsmustern. Weiterhin sind Marktteilnehmer in diesem Katalog verzeichnet.
- Die *Makler*-Komponente (*broker*) handelt im Auftrag eines oder mehrerer Marktteilnehmer, um potentielle Vertragsteilnehmer zu finden.
- Die *Verhandlung* (*negotiation*) von Verträgen durch mehrere Marktteilnehmer stellt eine weitere Komponente von Cosmos dar. Die Herausforderung

liegt in der Bereitstellung eines gemeinsamen Bearbeitungsmechanismus (*collaborative editing*) strukturierter Dokumente (Verträge), in der Unterstützung und Durchsetzung verschiedener *Verhandlungsprotokolle* und in der Bereitstellung verschiedener *Zugangsmechanismen* zur Bearbeitung des Vertrags, die sich nach den Bedürfnissen der Teilnehmer richten.

- Das *Signieren* (*signing*) des elektronischen Vertrags folgt in COSMOS der erfolgreichen Verhandlung eines Vertrags. Um ein elektronisches Dokument juristisch gültig signieren zu können, ist neben Techniken wie digitalen Signaturen (*public-key*-System) eine für Menschen lesbare Aufbereitung des gesamten Dokuments (WYSIWYS, "'what you see is what you sign") nötig. Eine objektive Instanz (*trusted third party*), auch als *elektronischer Notar* bezeichnet, dient zum Gegenzeichnen des von allen Parteien unterzeichneten Vertrags.

- Die *Ausführung* der Vertrags (*contract execution*) ist durch die Spezifikation eines Vertrags-Workflows möglich. Dieses Thema wird ausführlich im folgenden Abschnitt behandelt.

Auch für die Steuerung der Verhandlungen durch Protokolle wurde der Einsatz von Prozeßmodellierungstechniken betrachtet ([Tu et al. 1999b, Tu et al. 2000a]), worauf an dieser Stelle aber nicht weiter eingegangen wird.

Der im folgenden Abschnitt vorgestellte Ansatz zur Ausführung von Verträgen basiert auf dem (statischen) objektorientierten Modell eines Vertrags aus COSMOS. Als informationsorientiertes Modell wird im restlichen Teil dieses Abschnitts näher auf das Vertragsmodell eingegangen.

Abbildung 5.14 zeigt das statische objektorientierte Modell eines elektronischen Vertrags in COSMOS als UML-Klassendiagramm. Die Darstellung wurde aus [Cosmos-Consortium 1999] übernommen und zur besseren Übersicht um Aspekte vereinfacht, die nur die hier nicht betrachteten Phasen Verhandlung und Signatur betreffen. Dazu gehören die Buchführung der Änderungen am Vertrag (*logging*), die Notarisierung, verschiedene Arten von Klauseln, Signaturen und eine Stellvertreter-Modellierung. Für das vollständige Vertragsmodell sei auf [Cosmos-Consortium 1999] verwiesen.

Das Vertragsmodell gliedert sich nach [Cosmos-Consortium 1999] in die Bereiche Teilnehmer (*who*), Ausführung (*how*), Vertragsgegenstände (*what*), sowie legale (*legal*) und administrative Aspekte wie ein Modifikationsprotokoll. Für die Ausführung des Vertrags sind die Aspekte *who*, *what* und selbstverständlich *how* von Bedeutung.

Es soll nun auf die Einzelheiten des Klassenmodells in Abbildung 5.14 eingegangen werden. Ein *zusammengesetzter Vertrag* (Klasse CompoundContract) bündelt mehrere inhaltlich oder legal verknüpfte Verträge (Contract) zu einem Objekt. Ein Vertrag besteht im vereinfachten Modell aus den drei genannten Teilen *who*, *how* und *what*, die in der Abbildung jeweils durch UML-Notizen (*notes*) bezeichnet sind.

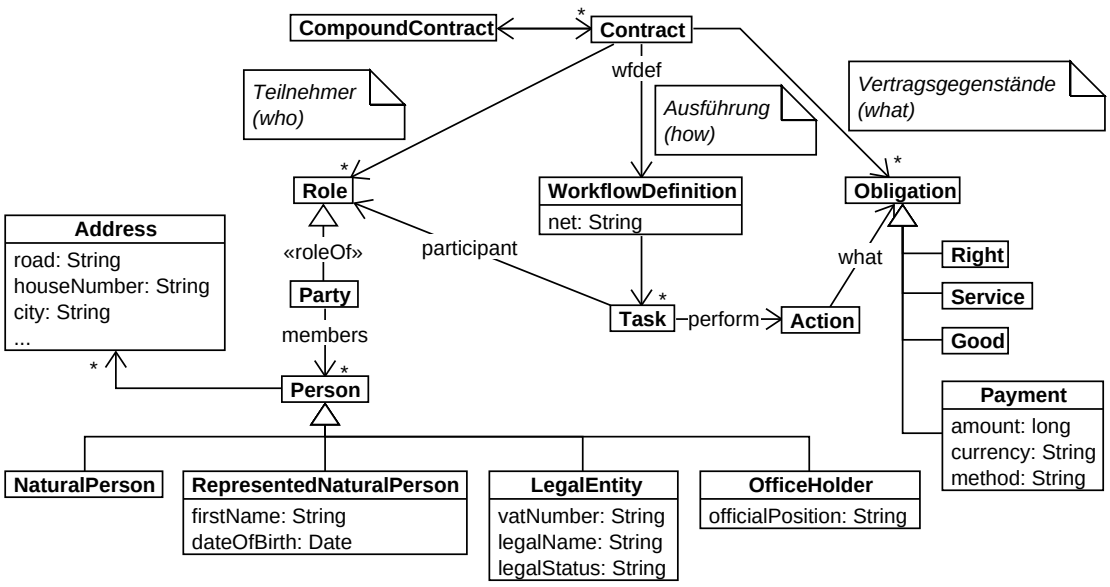

Abbildung 5.14: Das objektorientierte Vertragsmodell von COSMOS (vereinfacht).

Die *Teilnehmer* (Person) werden in *Vertragsparteien* (Party) organisiert. Jede Vertragspartei kann eine *Rolle* (Role) im Vertrag annehmen, wobei hier abweichend von [Cosmos-Consortium 1999] diese Beziehung als «roleOf»-Generalisierung, nicht als mehrwertige Assoziation modelliert wurde. Zur Diskussion von Rollen und Typen im Vergleich zu Assoziationen siehe Abschnitt 2.1.1 und [Hitz und Kappel 1999]. Zu jeder Person können beliebig viele Adressen (Address) mit den üblichen Attributen angegeben werden. Als Subklassen von Person werden verschiedene Arten von legalen oder natürlichen Personen modelliert. Der Unterschied erklärt sich im Zusammenhang mit Signaturen. Legale bzw. repräsentierte natürliche Personen müssen bzw. können über eine Vollmacht (*authority*) eine Stellvertreter (*proxy*) benennen.

Vertragsgegenstände (Obligation) werden durch die Subklassen *(Nutzungs-)Recht* (Right), *Dienst* (Service), *Ware* (Good) und *Zahlung* (Payment) spezialisiert. Bei der Zahlung sind Attribute für die Höhe der Zahlung (amount), die Währung (currency) und die Zahlungsmethode (method) vorgesehen.

Die *Ausführung* (WorkflowDefinition) ist hier nur grob angegeben, da diese im nächsten Abschnitt genauer behandelt wird. Man sieht, daß eine Workflow-Definition aus einer Anzahl von *Aufgaben* (Task) besteht, die einen für ihre Ausführung verantwortlichen Teilnehmer über eine Rolle bestimmen. Weiterhin gibt eine Aufgabe eine *Aktion* (Action) an, die wiederum bestimmt, auf welchen Vertragsgegenstand sich die Aufgabe bzw. die Aktion auswirkt.

5.2.2 Ausführung elektronischer Verträge in COSMOS

Nachdem im vorherigen Abschnitt der allgemeine Ansatz von COSMOS und das COSMOS-Vertragsmodell dargestellt wurden, soll nun genauer auf die Ausführung elektronischer Verträge eingegangen werden.

Die explizite Einführung eines elektronischen Vertrags stellt an sich bereits ein Novum des COSMOS-Ansatzes dar. In COSMOS leisten Verträge aber nicht nur einen Beitrag zur Formalisierung eines Dienstes und verbessern die juristischen Aspekte elektronischen Handels durch digitale Signaturen. Sie unterstützen zusätzlich die dritte Phase der Geschäftstransaktion, die Ausführung.

Wie oben bereits erwähnt ähnelt der ausführungsorientierte Teil der Spezifikation eines Vertrages einer Workflow-Definition. Die Unterschiede bestehen in der starken Integration von Vertragsobjekten, Rollen, juristischen Aspekten und der Ausführungsspezifikation durch das im vorherigen Abschnitt vorgestellte Vertragsmodell.

Eine ähnliche Verbindung von Vertrag und Workflow-Definition wird in [Grefen et al. 2000] vorgeschlagen. Im dort beschriebenen CROSSFLOW-Projekt näherte man sich dem Problem allerdings von der anderen Seite. Die Annahme ist, daß Teile der Geschäftsprozesse eines Unternehmen ausgelagert werden sollen. Im Fokus von CROSSFLOW steht damit die systemtechnische Unterstützung solcher *interorganisationaler Geschäftsprozesse*. Verträge werden hier verwendet, um die Art und Weise, wie der ausgelagerte Workflow vom beauftragten Unternehmen auszuführen ist, juristisch festzuhalten. Damit handelt es sich in CROSSFLOW um Verträge *über* Workflows, währen in COSMOS Verträge Workflows *enthalten*.

In COSMOS soll die Ausführung eines mit dem Vertragsmuster gegebenen und durch die Verhandlungsphase parametrisierten Vertrags-Workflow automatisch unterstützt werden. Dies ermöglicht zum einen eine genauere Kontrolle, ob die Vertragspartner die festgelegten Verpflichtungen erfüllen, zum anderen eine Teilnahme von Software-Agenten an der Vertragsausführung. Die Sichtweise von Agenten als Stellvertreter von Personen oder Parteien im E-Commerce wird in COSMOS auch in der Verhandlungsphase eingenommen, wie in [Tu et al. 1999a, Tu et al. 2000a, Tu et al. 2000b, Bartelt und Lamersdorf 2000] vorgestellt.

Abbildung 5.15 zeigt die in COSMOS gewählte Architektur zur Ausführung von Vertrags-Workflows. Dabei sind im Rahmen von COSMOS vollständig neu entwickelte Komponenten grau unterlegt dargestellt. Die anderen Komponenten wurden angepaßt oder wie vorhanden verwendet.

Das Teilmodell zur Spezifikation von Workflows in elektronischen Verträgen (COSMOS *Workflow-Definition*) wurde zunächst verfeinert. Das Ergebnis der detaillierteren Modellierung wird weiter unten vorgestellt. Als Workflow-Modell wurden *UML-Aktivitätsdiagramme* (siehe Abschnitt 3.1.3) als Vorbild herangezogen, wobei eine spezielle Abstimmung auf die COSMOS *Rollen und Obligationen* erfolgt ist. Zur graphischen Modellierung von Workflows wurde ein existierendes UML-Werkzeug

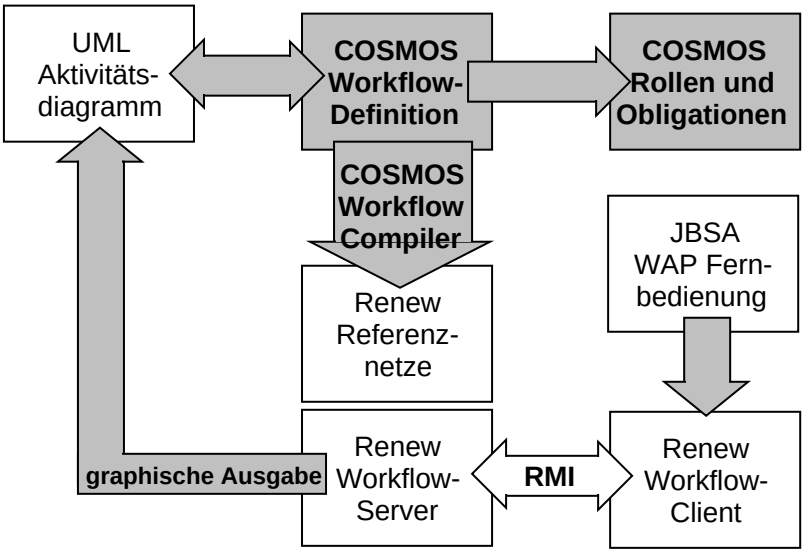

Abbildung 5.15: Architektur der Vertragsausführungs-Komponente von Cosmos.

angepaßt. Bei ArgoUML ([Robbins 1999, Tigris.org 2000]) handelt es sich um ein *open-source*-Projekt, so daß sämtliche Quellen zur Verfügung standen.

Als Workflow-Management-System wurde die in Abschnitt 3.5.2 beschriebene Workflow-Erweiterung von Renew verwendet. Diese läßt sich durch eine entsprechende API einsetzen, um automatisch generierte Workflows auszuführen. Durch die direkte Anbindung an Java (siehe Abschnitt 3.5) ist auch die Einbindung spezieller E-Commerce-Funktionen wie beispielsweise elektronischer Zahlungsverfahren möglich, da für diese Techniken meist Java-APIs zur Verfügung stehen.

Neben der Anpassung der graphischen Modellierung in ArgoUML stellt die Implementierung des Cosmos *Workflow-Compilers* die zentrale Aufgabe bei der Realisierung der Architektur dar, die vom Autoren durchgeführt wurde. Der Cosmos Workflow-Compiler bekommt als Eingabe eine Cosmos Workflow-Definition in dem im folgenden spezifizierten Format. Als Ergebnis wird dieses an Aktivitätsdiagrammen orientierte Format in Referenznetze mit Workflow-Erweiterungen (siehe Abschnitt 3.5.2) übersetzt.

Die generierten Workflow-Referenznetze können vom Renew-Workflow-Server ausgeführt und mit dem vorhandenen Workflow-Client über eine Java-RMI-Schnittstelle entfernt gesteuert werden (siehe Abschnitt 3.5.2).

Als besondere Anforderung von Cosmos wurde in Abschnitt 5.2.1 genannt, daß Benutzern abhängig von ihren Zugangsmöglichkeiten verschiedene Zugriffsmethoden auf die Cosmos-Infrastruktur ermöglicht werden soll. In [Müller-Wilken et al. 2000] wird die von Müller-Wilken entwickelte *Java Border Service Architecture* (JBSA,

[Müller-Wilken und Lamersdorf 2000]) zur Steuerung des Workflow-Clients durch Web-Browser, PDAs (*personal digital assistants*) und sogar durch WAP-fähige Mobiltelefone eingesetzt. Diese Lösung wurde auch im Rahmen von COSMOS erfolgreich für den Zugriff mobiler Benutzer auf Vertrags-Workflows eingesetzt.

Verfeinerung des Vertragsmodells bezüglich Ausführung

Das objektorientierte Vertragsmodell aus Abbildung 5.14 wurde im Bereich der Ausführung (*how*) verfeinert, um eine automatische Übersetzung in ausführbare Workflow-Definitionen zu erlauben.

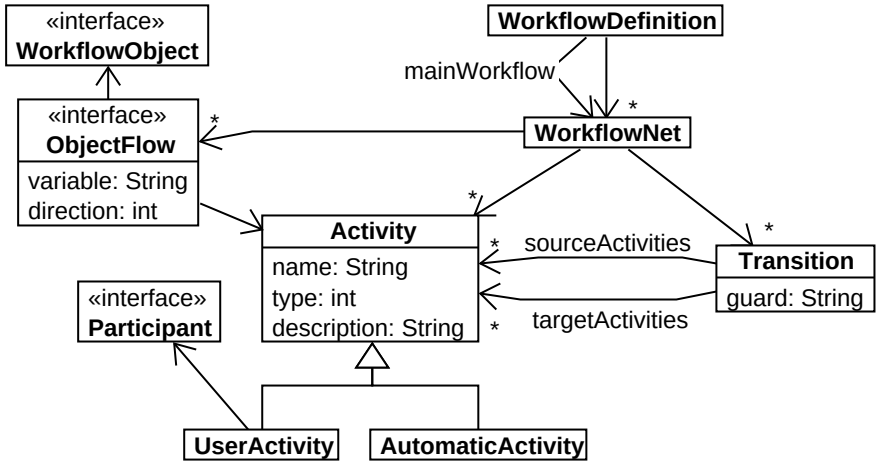

Abbildung 5.16: Abstraktes Entwurfsmodell der Workflow-Definition des COSMOS-Vertragsmodells aus Abbildung 5.14.

Abbildung 5.16 zeigt ein abstraktes Entwurfsmodell der entstandenen Klassenhierarchie. In der Implementierung wurden wie im COSMOS-Vertragsmodell alle Klassen als Schnittstellen *und* als Implementierungsklassen realisiert, um den Charakter eines Rahmenwerks (*framework*) zu betonen. Diese Trennung in Schnittstelle und Implementierungsklasse ist im Modell in Abbildung 5.16 nicht dargestellt. Weiterhin wurden die in Abschnitt 2.1.1 im Beispiel genannten Entwurfsmuster angenommen, was dazu führt, daß `get-`, `set-` und die anderen genannten Methoden durch entsprechende Attribute oder Assoziationen dargestellt werden.

Das Meta-Modell für COSMOS-Vertrags-Workflows aus Abbildung 5.16 stellt eine Vereinfachung und Anpassung eines Ausschnitts aus dem UML-Meta-Modell für Aktivitätsdiagramme dar. Eine WorkflowDefinition besteht aus einer Anzahl von Workflow-Netzen (WorkflowNet), von denen eines als der initiale Workflow (mainWorkflow) ausgezeichnet wird. Workflow-Netze können drei Arten von Objekten enthalten: Aktivitäten (Activity), Transitionen (Transition) und Objektflüsse

(ObjectFlow), die den Elementen eines UML-Aktivitätsmodells entsprechen.

Eine Transition besitzt einen Wächter (guard) und verbindet beliebig viele Eingangsaktivitäten (sourceActivities) mit beliebig vielen Ausgangsaktivitäten (targetActivities). Damit stellt eine Transition in Workflow-Netzen eine Verallgemeinerung der Steuerflußkante und des Synchronisationsbalken in UML-Aktivitätsdiagrammen dar (siehe Abschnitt 3.1.3).

Aktivitäten besitzen einen Namen (name), einen applikationsspezifischen Typ (type) und eine Beschreibung (description). Für Workflow-Netze werden zwei Subklassen von Aktivitäten betrachtet: benutzergesteuerte (UserActivity) und automatische Aktivitäten (AutomaticActivity). Hier sind leicht weitere Arten von Aktivitäten denkbar.

Benutzergesteuerte Aktivitäten spezifizieren zusätzlich einen Teilnehmer (Participant) und interpretieren die Beschreibung als natürlichsprachlichen Text. Der Teilnehmer ist in Abbildung 5.16 explizit als Schnittstelle angegeben, da diese zur höheren Flexibilität bei der Auswahl von möglichen Verantwortlichen für eine Aktivität von verschiedenen Klassen implementiert wird. Konkret können als Teilnehmer sowohl Rollen als auch Parteien oder einzelne Personen angegeben werden.

In automatischen Aktivitäten wird die Beschreibung als Code interpretiert. Durch die Möglichkeit von Renew, Netzanschriften mit Java-Code direkt auszuführen, können so automatische Aktivitäten durch beliebige Java-Implementierungen realisiert werden. Da Cosmos-Workflows keine expliziten Kantenanschriften tragen, werden vom Workflow-Compiler bestimmte Standardanschriften generiert, die in den Beschreibungen automatischer Aktivitäten genutzt werden können.

Um den Zugriff und die Modifikation von Vertragsobjekten vom Cosmos-Workflow aus zu erlauben, können Objektflüsse (ObjectFlow) angegeben werden (siehe Abschnitt 3.1.3). Ein Objektfluß ist genau einer Aktivität zugeordnet. Je nach der Belegung des Attributs direction ist der Objektfluß Vor- oder Nachbedingung der Aktivität. Durch Angabe einer Variablen (variable) kann in der Beschreibung einer automatischen Aktivität auf das Objekt aus dem Objektfluß Bezug genommen werden.

In Cosmos wird als Datenaustauschformat XML ([Oasis-Open 2000, Goldfarb und Prescod 2000]) eingesetzt. Das Vertrags-Objektmodell und das Workflow-Objektmodell können auf naheliegende Weise in eine XML-DTD überführt werden. Die in Java implementierten Klassen verwenden eine XML-DOM-API ([Sun 2000c]), um einen Objektgraphen als XML zu speichern bzw. aus einer XML-Repräsentation einen Objektgraphen zu erzeugen. Die DTD für das Workflow-Objektmodell ist in [Cosmos-Consortium 1999] spezifiziert.

Es wurde somit ein Objektmodell spezifiziert, das als Eingabe für den Cosmos-Workflow-Compiler dient.

Übersetzung von COSMOS-Workflows in Referenznetze

Der COSMOS-Workflow-Compiler liest einen Vertrags-Objektgraphen inklusive der Workflow-Objekte aus einer XML-Datei und erzeugt über die sogenannte *Shadow*-API von RENEW (siehe [Kummer et al. 2001]) ein entsprechendes Referenznetz als serialisierten Java-Objektgraphen.

Auf die Details der Übersetzung von COSMOS-Workflows in Referenznetze kann hier nicht näher eingegangen werden. Anhand der Darstellungen in Kapitel 3 sollte der enge Bezug zwischen Aktivitätsdiagrammen und Petrinetzen klar geworden sein. In den RENEW-Workflow-Erweiterungen (siehe Abschnitt 3.5.2) werden Transitionen mit speziellen Zielkanälen als Benutzer-Aktivitäten interpretiert. Steuerflußkanten werden durch Hilfstransitionen und -stellen dargestellt. Synchronisationsbalken können direkt mitsamt ihren Steuerflußkanten als eine Transition realisiert werden. Entscheidungsknoten entsprechen Stellen, die mehrere Transitionen im Nachbereich aufweisen.

Das vom Workflow-Compiler erzeugte serialisierte Referenznetz kann in RENEW geöffnet und bearbeitet oder simuliert werden. Alternativ kann das Referenznetz von der RENEW-Workflow-Engine geladen und instantiiert werden. Sodann kann sich ein Workflow-Client bei der Workflow-Engine über das Netzwerk anmelden und wie in Abschnitt 3.5.2 beschrieben Aufgaben anfordern und als erledigt melden.

5.2.3 Modellierung von Vertrags-Workflows mit FSNets

Referenznetze bieten durch ihre Java-Schnittstelle eine gute Integration in eine E-Commerce-Umgebung. Dennoch eignet sich Java als Anschriftensprache für Wächter und automatische Aktivitäten nur bedingt, wenn Fachexperten ohne Programmierkenntnisse die Vertrags-Workflows spezifizieren sollen. Als eine Alternative und ein weiteres Anwendungsbeispiel von FSNets wird in diesem Abschnitt ein Vertrags-Workflow als FSNet modelliert. Dabei wird ähnlich wie in Abschnitt 5.1 vorgegangen.

In Abschnitt 5.1 wurde ein Workflow-Meta-Modell vorgestellt, das hier leicht modifiziert wiederverwendet werden soll. Im folgenden werden gleichfalls Rollen und Aufgaben modelliert. Im Unterschied zu dem Workflow-Meta-Modell aus Abbildung 5.1 wird aber nicht der *Zustand* von Geschäftsobjekten modelliert, sondern die Geschäftsobjekte selbst. Dafür wird der Typ BOState durch einen Typ BO (*business object*) ersetzt.

Abbildung 5.1 zeigt eine Variante des Aktivitätsdiagramms aus Abbildung 3.2 als Vertrags-Workflow. Da in Vertrags-Workflows die Bestellung bereits mit dem Schließen des Vertrags durchgeführt wurde, gehört diese nicht in den Ausführungsteil. Die nebenläufigen Aktivitäten auf Seiten des Händlers wurden vereinfachend zu jeweils einem Schritt zusammengefaßt. Der Vertrags-Workflow nutzt *Rollen* und *Objektflüsse* (siehe Abschnitt 3.1.3). Damit der Händler aktiv wird, muß eine Be-

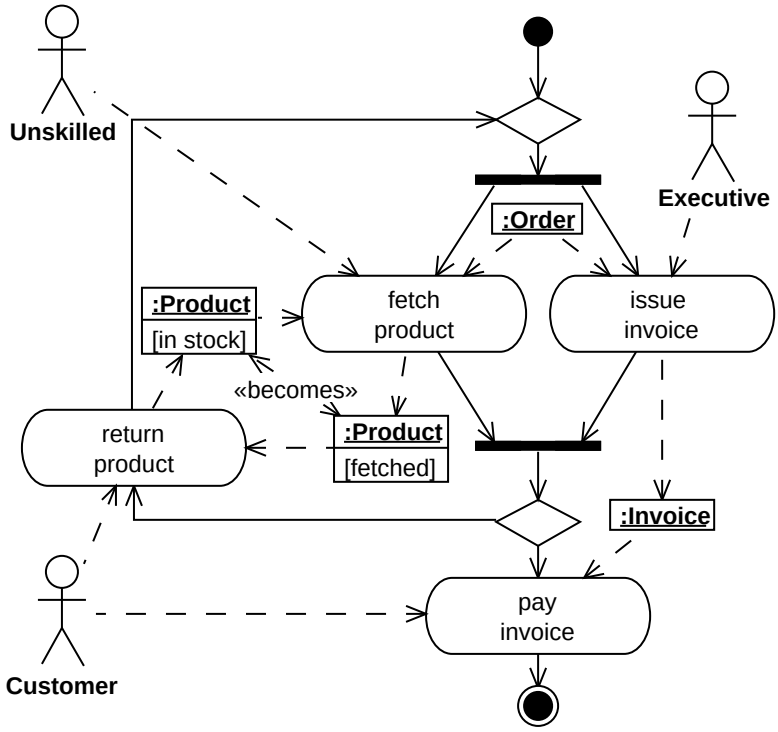

Abbildung 5.17: Auslieferung und Bezahlung eines Produkts als Vertrags-Workflow.

stellung (Order) vorliegen. Die Aushilfe (Rolle Unskilled) holt ein Produkt aus dem Lager (Aktivität fetch product), während der Sachbearbeiter (Rolle Executive) die Rechnung (Invoice) erstellt (issue invoice). Das eigentliche Verschicken von Produkt und Rechnung ist hier nicht explizit modelliert. Damit der Kunde (Rolle Customer) die Rechnung bezahlen kann (pay invoice), muß diese vorliegen. Um das Produkt zurückzugeben (return product) muß dieses beim Kunden vorliegen. Die Entscheidung zwischen diesen beiden Aktivitäten hängt davon ab, welche Aktivität der Kunde ausführt und wird nicht als explizite Entscheidungsaktivität modelliert. Eine solche Modellierung wird in [Aalst 1997] *implizite Entscheidung* (*implicit choice*) genannt.

Bei der Modellierung dieses Vertrags-Workflows als FSNet wird wiederum zuerst das Informations- und dann das Prozeßmodell erstellt. Das Informationsmodell ist genau genommen eine Erweiterung des Workflow-Objektmodells von COSMOS, wird hier aber auf der Basis des oben eingeführten Workflow-Meta-Modells spezifiziert.

Abbildung 5.18 zeigt das Typsystem, in dem die oben beschriebenen Rollen, Aufgaben und Geschäftsobjekte als Typen modelliert sind.

Die oben angegeben Rollen werden wie in Abschnitt 5.1 als Subtypen von Role

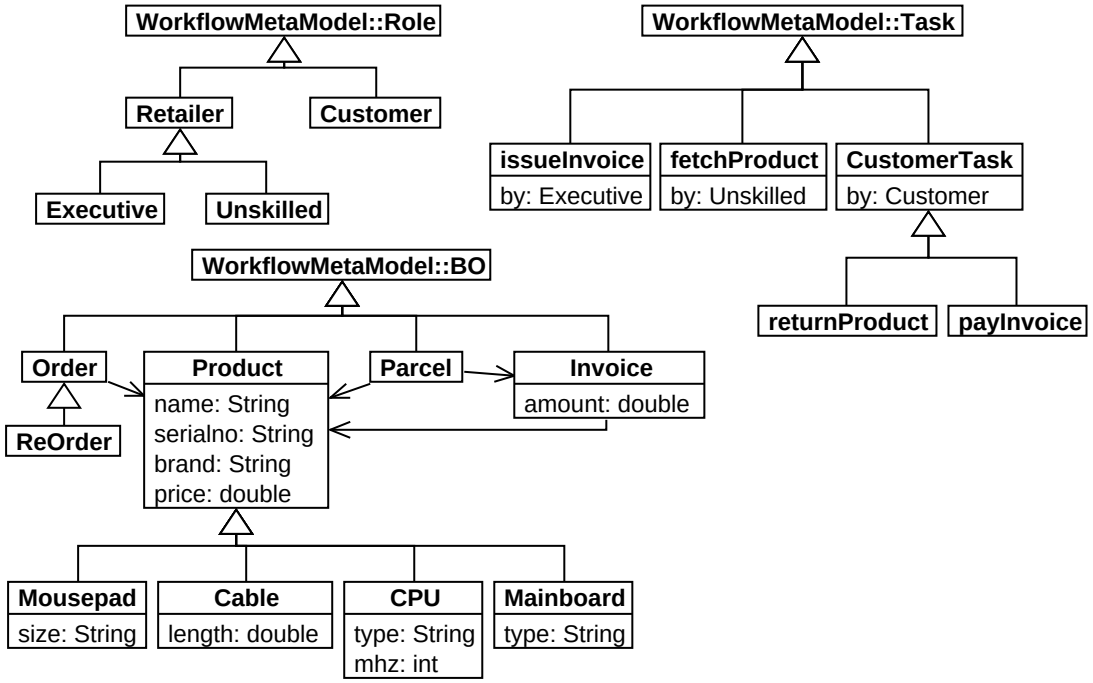

Abbildung 5.18: Das aus dem Vertrags-Workflow in Abbildung 5.17 abgeleitete Typsystem.

modelliert. Anhand von Executive und Unskilled wird gezeigt, daß Rollen in einem Typmodell wie in UML hierarchisch zu abstrakten Rollen wie Retailer zusammengefaßt werden können.

Die Aufgaben werden wie oben durch entsprechende Subtypen von Task modelliert, welche die verlangte Rolle weiter einschränken können. Eine Modellierung der erforderlichen *Zustände* von Geschäftsobjekten durch das Typsystem wird hier nicht vorgenommen.

Die Produkte werden als Geschäftsobjekte im wesentlichen wie im Beispiel aus Abschnitt 5.1 modelliert. Statt eines Lieferscheins gibt es in diesem Beispiel eine Rechnung (Invoice), die einen Rechnungsbetrag (amount) enthält. Produkt und Rechnung werden zum Versand in einem Paket (Parcel) zusammengefaßt. Der Typ ReOrder dient für die Rückgabe und Neubestellung eines Produkts.

Abbildung 5.19 zeigt den Vertrags-Workflow zur Auslieferung und Bezahlung eines Produkts aus Abbildung 5.17 als HFSNet. Die grobe Struktur des Aktivitätsdiagramms läßt sich leicht wiedererkennen. Im FSNet können allerdings einige Details der Informationsverarbeitung genauer modelliert werden.

Aktivitäten werden wie im Beispiel aus Abschnitt 5.1 als Transitionen durch einen Zielkanal mit einem Subtyp von Task modelliert. Durch die Features in und out werden in diesem Modell nicht nur wie oben die Marken im Zielkanal zur Verfügung

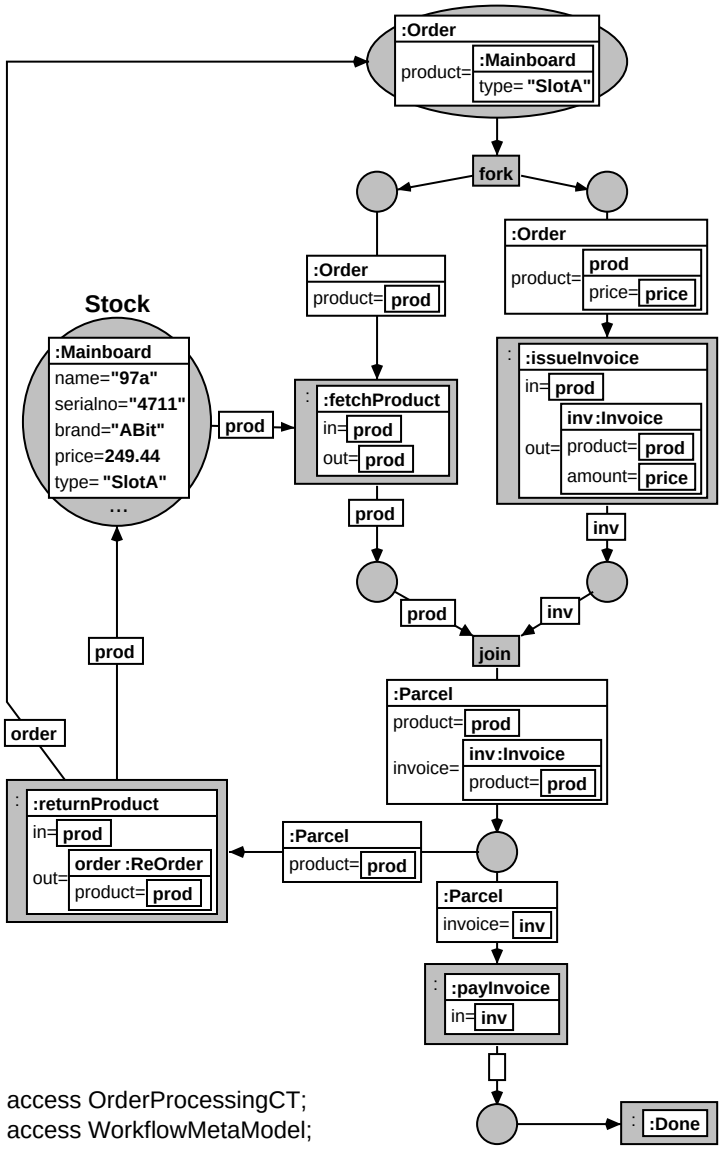

Abbildung 5.19: Vertrags-Workflow zur Auslieferung und Bezahlung eines Produkts als FSNet.

gestellt, sondern zusätzlich Unterstrukturen der Geschäftsobjekte selektiert oder gleichgesetzt. Auf einzelne Transitionsregeln wird unten genauer eingegangen.

Die Initialmarkierung des FSNets in Abbildung 5.19 beschreibt, daß der Vertrags-Workflow mit einem Geschäftsobjekt startet, das die Bestellung beschreibt. In diesem Fall wird ein Mainboard vom Typ SlotA bestellt. Durch einige Marken in der

Stelle Stock, von denen nur eine ausführlich dargestellt ist, wird die Auswahl der lieferbaren Produkte angegeben.

Man kann sich vorstellen, daß Vertrags-Workflows als FSNets in den oben beschriebenen Vertragsmustern von COSMOS auftreten und sich die Anfangsmarkierung (zumindest zum Teil) aus der Verhandlungsphase ergibt.

Die Transition fork erstellt zwei Kopien der Bestellung, die von den folgenden Transitionen mit den Zielkanälen fetchProduct und issueInvoice unabhängig verarbeitet werden können.

issueInvoice selektiert durch eine entsprechende Kantenanschrift das Produkt und dessen Preis aus der Bestellung und stellt das Produkt über den Zielkanal zur Verfügung (Feature in). Als Vorgabe für das Ergebnis der Transition (Feature out) wird eine Rechnung erzeugt, die auf das Produkt verweist und dessen Preis als Rechnungsbetrag enthält. Ein entsprechender Startkanal, der wie in Abschnitt 5.1 entweder vom Workflow-Client oder von einem simulierenden Netz ausgelöst wird, kann für das Ergebnis noch speziellere Information beisteuern.

fetchProduct selektiert ein Produkt aus dem Lager (Stock), das auf die in der Bestellung enthaltene Spezifikation paßt. Durch den Zielkanal kann wie zuvor die Entscheidung der Markenbindung von außen beeinflußt werden. Es ist auch möglich, dabei die Produktinformation noch weiter zu spezialisieren, in diesem Fall aber nicht unbedingt sinnvoll. Die Stelle Stock enthält im Normalfall bereits vollständig qualifizierte Produkte.

Die Transition join modelliert den Versand des Produkts gemeinsam mit der Rechnung als ein Paket. Durch Unifikation wird der erst jetzt konkret zur Verfügung stehende Preis in die Rechnung eingesetzt.

Die Transitionen mit den Zielkanälen returnProdukt und payInvoice stehen nun im Konflikt um die Marke, die das Paket repräsentiert. Der Konflikt wird dadurch entschieden, für welchen der Zielkanäle ein entsprechender Startkanal zur Verfügung gestellt wird. Wird der Workflow-Client benutzt, so entscheidet dies der Workflow-Teilnehmer. In einer Simulation hängt die Entscheidung von den im Umgebungsnetz zur Verfügung gestellten Ressourcen ab und wird im Zweifelsfall nichtdeterministisch getroffen. Die Details der beiden Transitionen ähneln den zuvor gezeigten Fällen und werden hier nicht näher dargelegt.

Die Transition mit dem Zielkanal Done dient wie in Abschnitt 5.1 dazu, die Beendigung des Workflows zu signalisieren und kann wie oben durch das Umgebungsnetz abgefragt werden.

COSMOS-Vertrags-Workflows können bisher nicht automatisch in entsprechende FSNets übersetzt werden, da der oben beschriebene Workflow-Compiler auf Referenznetze spezialisiert ist. Die Werkzeugunterstützung für FSNets war zum entsprechenden Zeitpunkt des Projekts noch nicht vollendet. Mit der FSNet-Modellierung wird eine verbesserte Modellierung von Geschäftsobjekten in Vertrags-Workflows möglich, die es auch fachlichen Experten ohne Programmierkenntnisse erlaubt, die informationsorientierten Aspekte darzustellen. Dies betrifft sowohl die Modellierung

der Geschäftsobjekte selbst, als auch die Spezifikation von Bedingungen und Regeln zu deren Verarbeitung.

Der existierende Workflow-Compiler könnte so erweitert werden, daß FSNets generiert werden können. Eine Alternative stellt dar, FSNets direkt als Modellierungstechnik für Vertrags-Workflows einzusetzen. Die UML-artige Notation von Feature Structures ist hier besonders geeignet, da UML-Kenntnisse von einem Modellierer erwartet werden können. Falls auch für die Prozeßstruktur eine UML-Notation erwünscht ist, kann auch der Weg beschritten werden, statt des externen Workflow-Compilers die Bearbeitung der Aktivitätsdiagramme direkt in das Modellierungswerkzeug FSRenew zu integrieren und intern auf Petrinetzstrukturen abzubilden.

Der letzte exemplarische Einsatz von FSNets stammt aus dem Gebiet der intelligenten Agenten. Da auf Agenten in dieser Arbeit bisher erst wenig eingegangen wurde, beginnt der Abschnitt mit einem Überblick und einer Einführung in Agenten.

5.3 Modellierung und Ausführung intelligenter Agenten

. . . you act,
and you know why you act,
but you don't know why you know that you know what you do.
Umberto Eco

Wie in der Einleitung dieser Arbeit erwähnt wird, ist der Begriff des „Agenten" in der Informatik nach wie vor nicht hinreichend genau definiert. Einer der Gründe hierfür ist, daß das Agenten-Konzept in verschiedenen Teildisziplinen der Informatik geprägt wurde: der KI und der (objektorientierten) Softwaretechnik. Dabei weichen die Auffassungen von Agenten nicht grundlegend voneinander ab, es gibt aber unterschiedliche Schwerpunkte.

In Abschnitt 5.3.1 wird ein kurzer Überblick über Agenten gegeben, wobei auf Einflüsse aus der KI und aus der Softwaretechnik eingegangen wird. Abschnitt 5.3.2 stellt den Ansatz der BDI-Agenten vor, der eine spezielle Variante intelligenter Agenten darstellt. Eine generische BDI-Agentenarchitektur wird in Abschnitt 5.3.3 als FSNet modelliert. Der Teil schließt mit einem Anwendungsbeispiel eines Abfall sammelnden BDI-Agenten, der vollständig als FSNet modelliert und ausgeführt werden kann.

5.3.1 Software-Agenten, Actors und verteilte Künstliche Intelligenz

Intelligente Agenten stellen eines der neueren Paradigmen der Künstlichen Intelligenz (KI) dar, das ein weites Anwendungsgebiet für sich erobern konnte. Man spricht im Zusammenhang mit mobilen Agenten oder anderen Anwendungen von KI im Bereich verteilter Systeme auch von *verteilter Künstlicher Intelligenz* (VKI). Man trifft auf den Begriff des Agenten sowohl bei Internet-Anwendungen, bei Steuersystemen

in der Fertigungsindustrie, bei Flugleitsystemen, bei militärischen Anwendungen als auch bei Spielen. Dabei ist der Begriff „Agent“ mit Vorsicht zu genießen, denn es existiert im Gegensatz zum „Objekt“ der Objektorientierung immer noch wenig Einigkeit darüber, was einen Agenten als solchen auszeichnet. Statt sich allgemeingültiger Definition anzumaßen, klassifiziert [Müller 1993] verschiedene in der Literatur zu findende Agenten nach den Kriterien Intelligenz, Anzahl und Autonomie. An Stelle von Autonomie ist in anderen Taxonomien ([Wooldridge und Jennings 1995]) auch der Begriff Verteilung anzutreffen. Daß Agentensysteme verteilte Systeme darstellen, ist durch die im allgemeinen asynchrone, nachrichtenbasierte Kommunikation zwischen Agenten gegeben.

Actors basieren auf den Arbeiten von Hewitt ([Hewitt et al. 1973]) und stellen eine sehr einfache, dafür aber formal beherrschbare Version von Agenten dar. Actors sind Objekten sehr ähnlich: Sie haben eine eindeutige Identität, kommunizieren ausschließlich über Nachrichten und kapseln so die in ihnen vorhandene Information. Der Unterschied ist, daß Actors über Regeln definiert werden, die ihr Verhalten deklarativ beschreiben. Zum Umfang der ersten Actor-Programmiersprache gehörte nur das Verschicken von Nachrichten sowie das Ändern des eigenen Verhaltens. Dabei sind Actors so einfach aufgebaut, daß sie nur aus einer endlichen Anzahl unterschiedlicher Verhaltensweisen auswählen können, die im voraus spezifiziert sein müssen.

Da Actors ereignisbasiert und zueinander nebenläufig arbeiten, wurden zu ihrer Beschreibung Petrinetze unabhängig von objektorientierten Petrinetzformalismen eingesetzt, zum Beispiel [Buchs und Guelfi 1991b] und [Lilius 1996]. Man kann sogar den Ursprung etlicher OO-Petrinetz-Ansätze in solchen Actor-Petrinetzen sehen, wie beispielsweise in mehreren Beiträgen in [Jensen und Rozenberg 1991].

Zur Beschreibung von Actors reichen die bekannten gefärbten Petrinetze vollkommen aus. Actors dienen aber eher zur Veranschaulichung der grundlegenden Konzepte als zur Spezifikation praxisrelevanter Systeme. Hier fehlt also eine Modellierungstechnik, welche sowohl die Ereignisbasierung und Nebenläufigkeit von Actors als auch die Dynamik intelligenter Agenten zu beschreiben vermag. Intelligenten Agenten gehen insofern über Actors hinaus, als bei diesen die Regeln dynamisch änderbar oder austauschbar sind. Dadurch werden Intelligente Agenten zu selbstkonfigurierenden Systemen. Tu widmet sich in [Tu et al. 2000a] der Steuerung intelligenter Agenten und zeigt, daß verteilte Systeme und E-Commerce lohnende Anwendungsgebiete für dieses neue Paradigma sind. Eine Modellierungstechnik für verteilte Systeme sollte demnach auch regelbasierte Modelle mit dynamisch austauschbaren Regeln unterstützen.

Im Gebiet der *intelligenten* Agenten, die auf diese grundlegenden Konzepte aufbauen, wurden bisher nur selten Petrinetze eingesetzt. Der Grund dafür scheint zu sein, daß Petrinetze die für komplexere Agenten notwendig werdende Informationsmodellierung bisher nur unzureichend unterstützen. [Wienberg 1996] stellt einen der ersten Versuche dar, mit CPN intelligente Agenten zu modellieren. Dazu wird ein

Ansatz ähnlich den Prolog-Netzen (siehe Abschnitt 3.4.4) verfolgt, der eine Kombination von Petrinetzen und Logikprogrammierung erlaubt. Weiterhin wird für die Agentennetze eine objektorientierte Petrinetz-Struktur nach [Moldt 1996] genutzt. Dieser Ansatz wurde in [Moldt und Wienberg 1997] und [Rölke 1999] weiterverfolgt.

FSNets benutzen mit Feature Structures eine Informationsmodellierungstechnik, die über reine Prädikatenlogik in einer Weise hinausgeht (siehe Abschnitt 2.3.1), die für viele KI-Problemstellungen eine elegantere Modellierung erlaubt. Vor allem im Bereich der Wissensrepräsentation, der für Agenten von zentraler Bedeutung ist, können Feature Structures ihre Vorteile gegenüber der Prädikatenlogik ausspielen. Dieser Abschnitt soll die Informationsmodellierung auf die hier verwendeten Typsysteme und Feature Structures übertragen, um dann auch die dynamischen Aspekte von BDI-Agenten durch FSNets formal zu beschreiben. Das FSNet modelliert also zunächst die allgemeine Architektur und Dynamik eines BDI-Agenten. Zur Veranschaulichung wird das in [Rao und Georgeff 1991] gegebene Beispiel eines Abfallsammler-Agenten als Anwendung mit FSNets modelliert. Dafür wird das allgemeine Modell mit entsprechenden Feature-Structure-Regeln ausgestattet sowie um die in der physikalischen Welt stattfindenden Aktionen ergänzt.

5.3.2 Einführung in BDI-Agenten

In diesem Abschnitt beziehen wir uns auf eine spezielle Art der intelligenten Agenten, die nach Rao, Georgeff und Kinny als BDI-Agenten (*belief, desire, intention*, [Rao 1996, Kinny et al. 1996]) bezeichnet werden. Der Ansatz geht zurück auf die Arbeit von Shoham, der mit [Shoham 1990] einen der ersten Beiträge zum Thema Agenten in der KI leistete und in [Shoham 1993, Shoham 1997] weiter verfolgt wird. Für die Erstellung eines konkreten Beispiels wurde angestrebt, auf einen wohldefinierten und damit speziellen Ansatz zurückzugreifen. BDI-Agenten setzen die wichtigsten allgemeinen Konzepte aus dem Bereich intelligenter Agenten um, so daß das Beispiel stellvertretend für allgemeinere Ansätze zeigt, daß intelligente Agenten mit FSNets modelliert werden können.

In [Rao und Georgeff 1991] und [Georgeff und Rao 1998] beschreiben die Autoren sehr detailliert, auf welchen Konzepten ein BDI-Agent basiert, wie seine interne Architektur aufgebaut ist und nach welchen Abläufen er gesteuert wird. Wie bei Shoham werden die Agentensprache und die Wissensrepräsentation recht formal beschrieben, bei der Architektur und dem dynamischen Verhalten wird hingegen meist auf Schaubilder und natürlichsprachliche Spezifikationen zurückgegriffen. Gegenüber existierenden mathematischen Verhaltensbeschreibungen wie z.B. *possible worlds semantics* (siehe z.B. [Ginsberg 1993]) haben Petrinetze den Vorteil der Ausführbarkeit.

Ein BDI-Agent kommuniziert mit der Umwelt durch Nachrichten. Dabei wird angenommen, daß bestimmte Ereignisse in der Welt von sogenannten *Sensoren* in Nachrichten umgesetzt werden, während die vom Agenten abgesandten Nachrichten

durch seine *Aktuatoren* in Aktionen umgesetzt werden können, welche die Welt beeinflussen bzw. verändern. Dadurch begrenzt sich die konzeptuelle Sicht auf die Interaktion eines Agenten mit seiner Umwelt wie bei Actors auf das Empfangen und Versenden von Nachrichten.

Die interne Architektur eines BDI-Agenten sieht vor, daß er über *subjektives Wissen* (*beliefs*), *Wünsche* oder auch *Ziele* (*desires*) und Absichten (*intentions*) verfügt. Das Wissen wird in einer lokale Wissensbasis umgesetzt, während sich die Ziele in Form von *Plänen* (*plans*) manifestieren, die beschreiben, welche Absichten durch ein Ereignis in einem bestimmten Wissenskontext generiert werden. Dazu geben Rao und Georgeff die Struktur eines Planes recht genau an. Ein Plan enthält demnach ein Muster für das auslösende Ereignis (*trigger*), einen damit verknüpften Kontext der lokalen Wissensbasis (*context*) und eine Absicht (*intention*), die sich auf beide andere Komponenten beziehen kann und aus einer Abfolge von Absichtsschritten (*intention steps*) besteht. Ein solcher Schritt kann das Versenden einer Nachricht, der Aufruf einer internen Funktion oder eine atomare Aktion sein, welche direkt einen Aktuator anspricht.

Ein BDI-Agent besitzt eine Ereigniswarteschlange, in die eintreffende Nachrichten aufgenommen werden. Parallel dazu werden die Ereignisse verarbeitet, wozu ein passender Plan herangezogen wird. Damit ein Plan „passend“ ist, müssen folgende Bedingungen erfüllt sein:

- Das auslösende Ereignis muß auf das Ereignismuster des Plans passen.
- Der im Plan geforderte und üblicherweise von Ereignisparametern abhängige Kontext muß aus der aktuellen Wissensbasis ableitbar sein.
- Die Absicht muß in der im Plan angegebenen Form mit den durch das Ereignis und den Kontext gegebenen Werten instanziierbar sein.

Wird ein Plan angewandt, wird das auslösende Ereignis konsumiert und die Absicht in Form der instanziierten Liste von Schritten abgelegt. Eine weitere nebenläufige Komponente ist für die Abarbeitung aller aktiven Absichten zuständig. Dafür wird der erste Schritt einer aktiven Absicht ausgeführt, indem die Nachricht versandt, die Funktion aufgerufen oder der Aktuator aktiviert wird. Schritte können insbesondere auch Nachrichten an den Agenten selbst sein, die beispielsweise zur Aktivierung eines weiteren Plans führen können.

5.3.3 Modellierung einer generischen BDI-Agenten-Architektur durch FSNets

Zunächst soll die allgemeine BDI-Agenten-Architektur als FSNet modelliert werden.

Die drei bereits von Shoham identifizierten, nebenläufig arbeitenden Funktionseinheiten eines Agenten werden als Transitionen modelliert, wogegen die Ereignisse

und Wissen als Feature Structures in Stellen abgelegt werden. Die drei Funktionseinheiten sind für das Empfangen von Nachrichten, das Verarbeiten von Nachrichten mit Hilfe von Plänen und das Ausführen der Absichten des Agenten verantwortlich. Daraus ergibt sich eine erste grobe Netzstruktur, die in Abbildung 5.20 zu sehen ist.

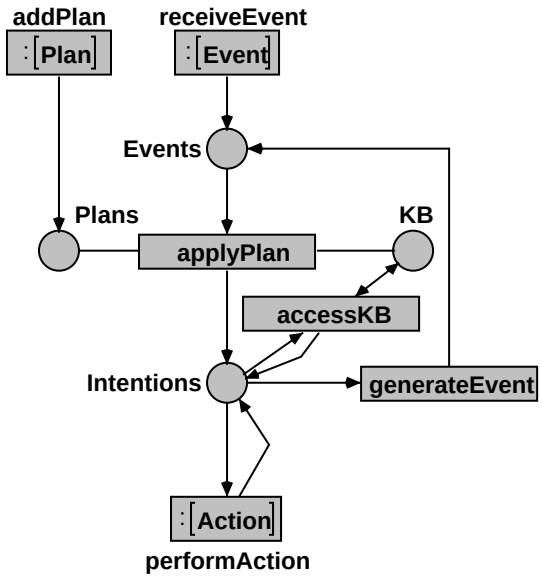

Abbildung 5.20: Die Architektur eines BDI-Agenten als schematisches FSNet.

An der Transition receiveEvent wird durch einen Zielkanal modelliert, daß diese Nachrichten von anderen Agenten (oder der „Welt“) übermittelt bekommt, die in der Stelle Events abgelegt werden. Ebenso können Pläne über einen Zielkanal hinzugefügt werden (addPlan).

Die Transition applyPlan wählt einen passenden Plan aus der Stelle Plans und greift zur Überprüfung des Kontext auf die Wissensbasis in der Stelle KB (*knowledge base*) zu. Es wird eine neue Absicht auf der Stelle Intentions abgelegt, die von der dritten Funktionseinheit, die sich aus den Transitionen performAction, accessKB und generateEvent zusammensetzt, elementweise abgearbeitet wird. Der Zielkanal :[Action] signalisiert, daß eine durch einen konkreten Agenten zu realisierende Aktion auszuführen ist.

Eine in der generischen Architektur vorgesehene Aktion ist, daß der Agent sich selbst ein Ereignis zusendet. Dies wird beispielsweise genutzt, um als eine Aktion einen weiteren Plan aktivieren zu können. Auch der Zugriff auf die eigene Wissensbasis könnte durch Nachrichten erfolgen, wurde hier aber der Einfachheit halber direkt modelliert. Da mit der Bearbeitung der Absicht erst fortgefahren werden soll, wenn die aus dem Unterplan resultierende Absicht abgearbeitet ist, wird in einem

solchen internen Ereignis die Liste der verbleibenden Schritte mitgeschickt.

Wie durch die Benutzung der synchronen Kanäle bereits angedeutet, wird zur Modellierung der BDI-Agenten wie bei der Workflow-Modellierung der HFSNet-Formalismus aus Abschnitt 4.4 aufgegriffen, der FSNets und Referenznetze verbindet. Damit ist eine Modellierung möglich, die einen Agenten aus mehreren kommunizierenden Teilobjekten zusammengesetzt darstellt. Doch bevor die einzelnen Netze, aus denen das Agentenmodell besteht, detailliert vorgestellt werden, soll zunächst das Typsystem hergeleitet werden.

Typsystem des generischen BDI-Agenten

In [Rao und Georgeff 1991] wird die oben beschriebene Struktur eines Plans durch eine Modallogik dargestellt. Diese Struktur muß nun auf ein Typ- bzw. Konzeptsystem übertragen werden. Abbildung 5.21 zeigt das resultierende Konzeptsystem in UML-Notation.

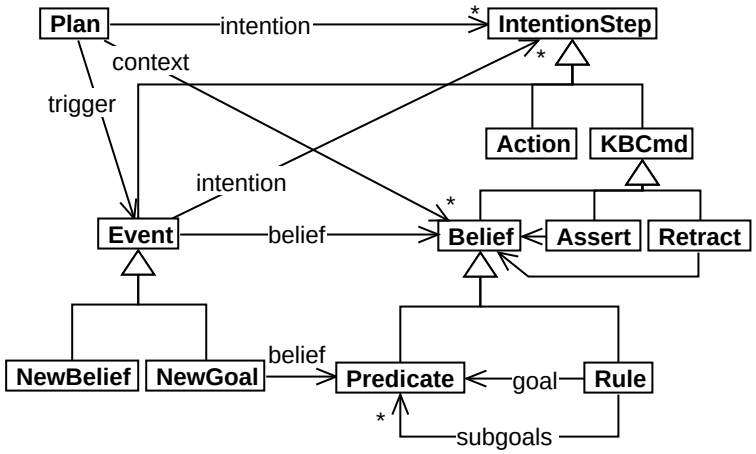

Abbildung 5.21: Das Typmodell zur BDI-Agenten-Architektur.

Ein Plan besteht aus einem Auslöser (Feature trigger vom Typ Event), einem Wissenskontext (Feature context vom Typ Belief*) und einer Absicht (Feature intention vom Typ IntentionStep$*$).

Im einfachsten BDI-Ansatz gibt es nur zwei Arten von Ereignissen, die sich beide auf ein Belief-Objekt (siehe unten) beziehen: Entweder, es wurde neues Wissen akquiriert (NewBelief), oder es wurde ein neues Ziel übernommen (NewGoal), das durch ein Prädikat beschrieben wird. An dieser Stelle wird im Modell von der Möglichkeit Gebrauch gemacht, den Typ eines Attributs bzw. einer Assoziation (in diesem Fall das als Assoziation dargestellte Feature belief) im Subtyp zu spezialisieren. Um in den oben erwähnten internen Ereignissen die Liste der restlichen Schritte ange-

ben zu können, enthalten auch Ereignisse eine Absicht (Feature intention vom Typ IntentionStep∗). Bei externen Ereignissen wird dafür gesorgt, daß dieses Feature mit einer leeren Liste belegt wird (siehe unten).

Ein Element des Wissenskontext (Belief) kann entweder ein Prädikat (Typ Predicate) oder eine Regel (Typ Rule) sein. Prädikate werden hier auch als *Fakten* bezeichnet. Eine Regel wird in Form einer Hornklausel aus einem Ziel (goal) und einer Menge von Unterzielen (subgoals) dargestellt. Die Unterziele müssen bewiesen werden, damit das Ziel selbst als bewiesen gilt. Ziel und Unterziele sind entsprechend vom Typ Predicate.

Ein Schritt einer Absicht (Typ IntentionStep) kann schließlich ein (internes) Ereignis, eine von einem konkreten Agenten durch einen Aktuator zu realisierende Aktion (Typ Action) oder eine Abfrage oder Änderung der Wissensbasis (Typ KBCmd, *knowledge base command*) sein. Zur Änderung der Wissensbasis stehen die aus PROLOG bekannten Befehle Assert (neues Prädikat oder neue Regel in die Wissenbasis aufnehmen) und Retract (Prädikat oder Regel aus der Wissensbasis entfernen) zur Verfügung (siehe [Shoham 1994, Sterling und Shapiro 1994]).

Prozeßmodell des generischen BDI-Agenten

Kehren wir zurück zur Prozeßmodellierung des BDI-Agenten. Die Architektur aus Abbildung 5.20 soll nun verfeinert werden. Da der Vorgang, Wissen aus Fakten und Regeln abzuleiten, selbst mit Feature Structures nicht trivial sein wird, bietet es sich an, die Wissensbasis als eine Komponente des Agenten zu modellieren.

Das FSNet, welches das Kernstück des Agenten darstellt und der oben gezeigten groben Architektur entspricht, soll im folgenden BDIAgent heißen. In diesem Netz werden zu empfangenen Ereignissen die passenden Pläne aktiviert und die Absichten des Agenten ausgeführt. Die eigentliche Ausführung der Aktionen wird sich in verschiedenen konkreten Agenten nach deren Fähigkeiten (Aktuatoren etc.) unterscheiden und wird deshalb vorerst nur als Schnittstelle betrachtet.

Abbildung 5.22 zeigt das verfeinerte BDI-Agenten-Modell aus Abbildung 5.20 als HFSNet. Die Symbole links oben in der Darstellung von diesem und den folgenden Netzen werden in RENEW der anschaulichen Darstellung von Netzreferenzen genutzt: Eine Netzreferenz kann statt durch den Namen des Basisnetzes (mit einem entsprechenden Index) durch dessen Piktogramm dargestellt werden.

Die Wissensbasis wird als eigenes Netz namens *KB* modelliert, von dem in der gleichnamigen Stelle durch die Java-Feature-Structureein Netzexemplar als Anfangsmarkierung erzeugt wird. Der Zugriff auf die Wissensbasis erfolgt an verschiedenen Stellen des Netzes und soll auch konkreten Agenten zur Verfügung stehen, weshalb er hier durch einen Zielkanal modelliert wird, der eine Anfrage mit einem Startkanal an die Wissensbasis weiterreicht (Transition *delegateKBCmd*).

Einem empfangenen externen Ereignis wird von der Transitionsregel von receiveEvent eine leere Liste von Absichtsschritten zugewiesen (siehe oben). Im vorliegen-

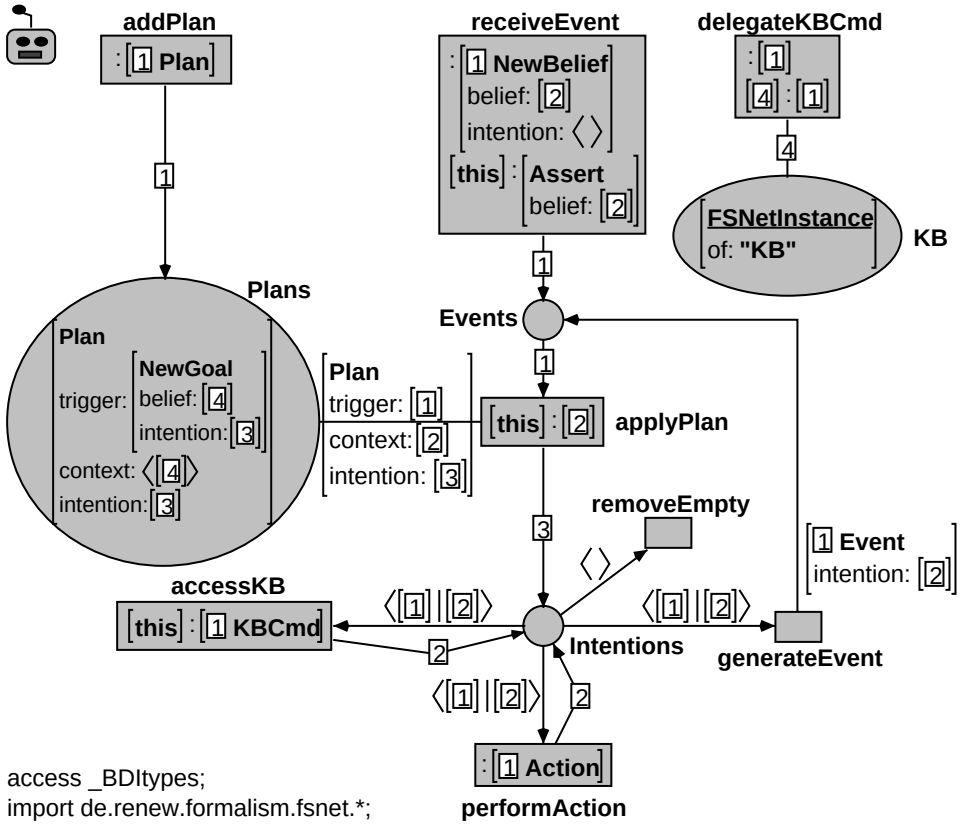

Abbildung 5.22: Das verfeinerte BDI-Agenten-Modell als HFSNet BDIAgent.

den Modell werden nur externe Ereignisse vom Typ NewBelief betrachtet, NewGoal-Ereignisse treten im unten folgenden konkreten Agenten nur als interne Ereignisse auf. Das eingetroffene Prädikat wird durch einen Assert-Startkanal sofort in die Wissensbasis aufgenommen. Das Ereignis wird in der Stelle Events abgelegt. Dort können sich mehrere Ereignisse ansammeln, bis sie zur Aktivierung eines Plans herangezogen werden.

Die Transition applyPlan wendet passende Pläne nach den oben spezifizierten Regeln an. Dazu wählt sie ein Ereignis und einen Plan aus, wobei der Pfadwert unter trigger (Knotenreferenz [1]) mit dem Ereignis unifizierbar sein muß. Weiterhin muß der eventuell durch diese erste Unifikation spezialisierte Kontext des Plans (Knotenreferenz [2]) durch die Wissensbasis beweisbar sein. Dies wird durch den Zielkanal [this] : [[2]] ausgedrückt. Können alle Unifikationen durchgeführt werden, so erhält man unter Knotenreferenz [3] die instantiierte Absicht, die in der Stelle Intentions abgelegt wird.

Der generische Agent ist anfangs mit einem einzigen Plan ausgestattet, der in

Abbildung 5.22 als Anfangsmarkierung der Stelle Plans zu sehen ist. Dieser Plan steht für den trivialen Fall, daß ein Ziel erreicht werden soll (Knotenreferenz 4 in NewGoal), das aus dem Kontext der Wissensbasis bereits bewiesen werden kann. Der Agent braucht dann keine Aktionen durchzuführen, um das Ziel zu erreichen. Als Absicht werden für den Fall, daß es sich um ein internes Ereignis handelt, die im Ereignis angegebenen Schritte übernommen (Knotenreferenz 3).

Die Transitionen accessKB, performAction und generateEvent stellen die oben genannten drei Möglichkeiten für einzelne Schritte einer Absicht dar. Jede der Transitionen betrachtet nur den Kopf einer aktuellen Absicht (auch hier können mehrere Marken und damit mehrere Absichten zur Auswahl stehen), Änderungen an den übrigen Schritten der Absicht können sich aber aus Koreferenzen ergeben. Welche der drei Transitionen für eine Absicht aktiviert ist, ergibt sich eindeutig aus dem Typ ihres ersten Schritts, da die Typen KBCmd, Action und Event wechselseitig inkompatibel sind.

Der Zugriff auf die Wissensbasis wird von accessKB einfach an diese delegiert. Eine Aktion wird von performAction über einen Zielkanal nach außen bekanntgegeben. Ein Ereignis wird von generate event als internes Ereignis auf die Stelle Events transportiert, wobei die Liste der restlichen Schritte unter das Feature intention gestellt wird (siehe oben).

Die Transition removeEmpty schließlich dient der Beseitigung von leeren Absichten (*garbage collection*) und ist damit für eine korrekte Funktion des generischen Agenten nicht unbedingt erforderlich.

Kommen wir nun zum Modell der Wissensbasis. Diese soll Fakten und Regeln speichern und einen Kontext beweisen können.

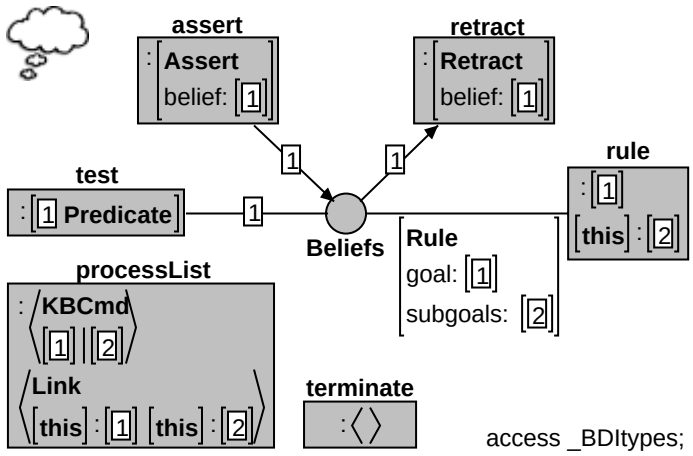

Abbildung 5.23: Die Wissensbasis eines BDI-Agenten als HFSNet KB.

Abbildung 5.23 zeigt das Modell einer Wissensbasis als HFSNet. Die Fakten und

Regeln werden in der Stelle Beliefs gespeichert. Über mit Assert bzw. Retract typisierte Zielkanäle können Prädikate und Regeln in die Wissenbasis aufgenommen bzw. aus ihr entfernt werden. Man beachte, daß für ein Retract in einem entsprechenden Startkanal nur eine Feature Structure angegeben werden muß, die mit einer Marke aus der Stelle Beliefs *unifiziert*. Dadurch können auch Fakten und Regeln entfernt (und dabei noch ein letztes Mal gelesen) werden, die nicht in allen Details bekannt sind.

Die Transition test bietet über einen Zielkanal die Möglichkeit, eine Anfrage mit einem Prädikat aus der Stelle Beliefs zu unifizieren, ohne deren Markierung zu ändern (Testkante).

Durch Transition rule wird die Anwendung einer Regel realisiert. Zu dem durch den Zielkanal gegebenen Prädikat (Knotenreferenz 1) wird eine Regel auf der Stelle Beliefs gesucht, deren Ziel (goal) mit dem Prädikat unifizierbar ist. Aus der Unifikation ergibt sich die Liste der zu beweisenden Unterziele (subgoals, Knotenreferenz 2), die durch einen Startkanal an die Transition processList desselben Netzes übergeben wird.

Transition processList verarbeitet eine Liste von zu beweisenden Prädikaten oder Regeln, indem sie die über den Zielkanal gegebene Liste in Kopf (Knotenreferenz 1) und Restliste (Knotenreferenz 2) zerlegt und Zielkanäle desselben Netzes *rekursiv* aufruft. Die beiden Zielkanäle können in FSRenew als Liste vom speziellen Typ Link notiert werden, um eine Auswertungsreihenfolge vorzugeben. Dies entspricht einem der prozeduralen Aspekte in Prolog (siehe [Sterling und Shapiro 1994]) und dient der Verhinderung von Endlosschleifen aufgrund ungünstiger Reihenfolgen bei Tiefensuche.

Die Transition terminate stellt das Rekursionsende der Listenbearbeitung dar. Eine leere Liste gilt generell als beweisbar.

Die Modellierung von Beweisen durch rekursive (sich zyklisch aufrufende) synchrone Kanäle ermöglicht eine kompakte Modellierung einer Wissensbasis, die eine nicht-triviale Komponente von Agenten darstellt. Eine Modellierung des Beweisvorgangs selbst als FSNet sollte ebenfalls möglich sein, wenn man dem in [Wienberg 1996] gezeigten Ansatz folgt. Dies ist aber nicht unser Ziel, da eine Agentenarchitektur, nicht ein Beweismechanismus modelliert werden soll.

Im folgenden Abschnitt werden die generischen BDI-Agenten-Netze in einem Beispiel eingesetzt, in dem ein konkreter Agent die Aufgabe hat, Abfall von einer mehrspurigen Straße einzusammeln.

5.3.4 Anwendungsbeispiel für einen BDI-Agenten: Abfallsammler

Im folgenden wird ein konkreter BDI-Agent auf der Basis der im vorherigen Abschnitt entwickelten generischen Architektur modelliert. Das Beispiel wird aus

[Rao und Georgeff 1991] aufgegriffen, wo ein Abfall sammelnder Roboter[1] modelliert wird.

Das Problem wird wie folgt beschrieben. Ein Roboter hat die Aufgabe, den auf drei angrenzenden Fahrbahnen a, b und c auftretenden Abfall einzusammeln und in einen Mülleimer zu befördern, der auf Fahrbahn b steht. Der Roboter kann sich von einer Fahrbahn zur einer angrenzenden bewegen und besitzt einen Greifer, mit dem er (abweichend von der Darstellung in [Rao und Georgeff 1991]) mehrere Abfälle einsammeln kann, bevor er diese im Mülleimer entsorgt.

Verschiedene Aspekte sollen an diesem Problem deutlich werden:

- Der Roboter bewegt sich in einer Welt, in der Ereignisse auftreten, die er nicht beeinflussen kann („Neuer Abfall auf Fahrbahn x“) und auf die er *reagieren* muß.
- Trotz der sehr einfachen Topologie soll der Roboter seine Bewegungen *planen*, um für Änderungen in der Welt vorbereitet zu sein (beispielsweise neue Fahrbahnen oder eine erweiterte Topologie, eine variierende Position des Mülleimers etc.).
- Die Regeln, nach denen der Roboter plant, sollen austauschbar sein, damit sich der Roboter (durch Lernen oder durch expliziten Eingriff von außen) an veränderte Anforderungen *anpassen* kann.

Wie in den vorhergehenden Modellierungsbeispielen wird zunächst das Informationsmodell und dann das Prozeßmodell aufgestellt, wobei beide Modelle Erweiterungen des generischen BDI-Agenten darstellen.

Informationsmodell des Abfallsammlers

Ein konkreter BDI-Agent benötigt ein Modell seiner Domäne. Im Beispiel des Abfallsammlers benötigt dieser Wissen über die Fahrbahnen und deren Topologie, über den Ort von Dingen wie Abfall, Mülleimer und nicht zuletzt seiner selbst. Weiterhin müssen die möglichen Aktionen, die der Agent über seine Aktuatoren auslösen kann, als Typen modelliert werden. Im Beispiel ergeben sich die Aktionen Bewegung, Aufheben und Entsorgen.

Abbildung 5.24 zeigt das Typmodell für den Abfallsammler-Agenten. Im unteren Teil wird ein Modell von Objekten, Entitäten und Orten angelegt, das für ähnliche Anwendungen wiederverwendet werden kann. Im einzelnen sind Objekte (Object) Entitäten (Entity) oder Orte (Location), wobei ein Behälter (Container) beides gleichzeitig ist. Die konkreteren Typen der Domäne des Beispiels sind der Abfall (Waste), der einen Namen bekommen kann, der Roboter selbst (robot) und der Mülleimer

[1] Trotz der Verwendung des Begriffs „Roboter“ steht bei der Modellierung die Planung und Ausführung von Aktionen im Vordergrund. Die Aktionen müssen nicht unbedingt aus physikalischen Handlungen bestehen. Es könnte sich ebenso gut um einen Software-Agenten handeln, der sich durch das Internet „bewegt“.

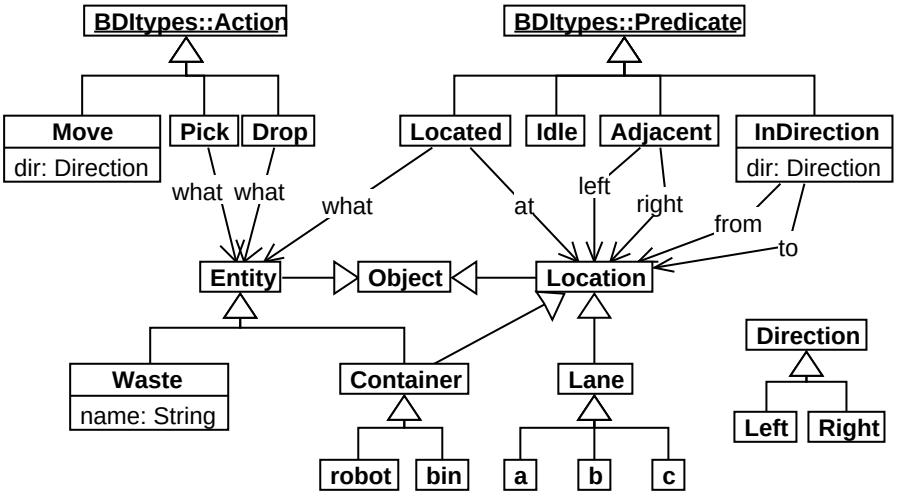

Abbildung 5.24: Die Erweiterung des BDI-Agenten-Typsystems für einen konkreten Agenten, der Abfall sammeln soll.

(bin), sowie die drei Fahrbahnen (Lane) a, b und c. Die klein geschriebenen Typen sind *extensional* zu verstehen, mit einem Knoten vom Typ robot ist demnach immer dasselbe Objekt gemeint. Damit stehen diese Typen für bestimmte Objekte.

Als Subtypen von BDItypes::Predicate werden verschiedene Prädikate vereinbart, die der Abfallsammler für seine Wissensrepräsentation benötigt. Located gibt über die durch Assoziationen notierten Features what und at eine Entität und einen Ort an, an dem sich diese befindet (zumindest glaubt dies der Agent). Das Prädikat Idle modelliert den Zustand des Agenten, daß er eine neue Bewegungsphase starten kann. Mit dem Prädikat Adjacent wird das Wissen des Agenten über die Topologien der Fahrbahnen repräsentiert, wobei in diesem einfachen Modell nur ein direktes Nebeneinanderliegen über die Features left und right angegeben werden kann. Das Prädikat InDirection dient schließlich der Planung einer Bewegungsphase von einer Startfahrbahn zu einer Zielfahrbahn.

Als Aktionen (Subtypen von BDItypes::Action) stehen dem Agenten Bewegen (Move), Abfall aufheben (Pick) und Abfall entsorgen (Drop) mit den offensichtlichen Features zur Verfügung. Sowohl für die Move-Aktion als auch für die Planung einer Bewegungsphase wird der Hilfstyp Direction benötigt, der die Subtypen Left und Right annehmen kann.

Im folgenden werden die initiale Wissensbasis und die Pläne des Agenten entwickelt, die im Prozeßmodell als Anfangsmarkierung verwendet werden. Der Entwurf des Wissens und der Pläne des Agenten stellt eine Informationsmodellierung dar, da diese weder als reine Daten, noch als eine prozedurale Beschreibung von Abläufen angesehen werden. Pläne sind eine statische Beschreibung, aber gleichzeitig eine

deklarative Spezifikation dynamischen Handelns.

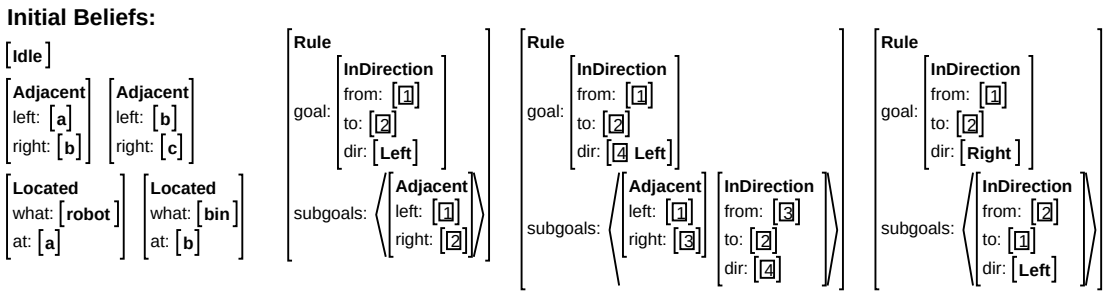

Abbildung 5.25: Das initiale Wissen des Abfallsammler-Agenten.

Abbildung 5.25 zeigt das initiale Wissen des Abfallsammler-Agenten. Als initiales Wissen wird dem Abfallsammler-Agenten die momentane Situation in seiner Welt mit auf den Weg gegeben. Die Feature Structure [Idle] besagt, daß der Roboter anfänglich keine Bewegungsphase gestartet hat. Die beiden Feature Structures vom Typ Adjacent beschreiben die Nachbarschaft der Fahrbahnen a und b sowie b und c. Die Located-Prädikate geben an, daß sich der Roboter zunächst auf Fahrbahn a und der Mülleimer auf Fahrbahn b befindet.

Ein BDI-Agent verfügt nicht nur über Pläne, die sein Handeln steuern, sondern auch über Regeln, die seine Wissenbasis über die Eigenschaften einer Datenbank hinaus Möglichkeiten zur Inferenz weiterer Information geben. Obwohl dies bei der primitiven Topologie des Beispiels nicht unbedingt nötig wäre, wird die Bewegungsrichtung hier durch Regeln bestimmt, die auf dem Wissen über die Topologie der Fahrbahnen aufbauen.

Die erste Regel besagt, daß ein Ort [1] links von einem anderen Ort [2] liegt, wenn [1] direkt links an [2] angrenzt. Die zweite Regel erweitert diesen Fall, indem von der direkten Nachbarschaft durch rekursiven Aufruf des InDirection-Prädikats auf die transitive Nachbarschaft mit einem Zwischenort [3] geschlossen wird. Die dritte Regel führt den Fall, daß sich ein Ort rechts von einem anderen liegt, auf den vorherigen Fall zurück, indem die Argumente left und right vertauscht werden.

Abbildung 5.26 zeigt die initialen Pläne des Abfallsammler-Agenten. Wie oben beschrieben gibt jeder Plan ein auslösendes Ereignis, einen Wissenskontext und eine Absicht an. Der Abfallsammler verfügt neben dem oben beschriebenen Standard-Plan nur über zwei weitere Pläne, einen Haupt- und einen Hilfsplan.

Der in der Abbildung oben dargestellte Plan wird aktiviert, sobald ein Ereignis eingetroffen ist, das den Agenten über neuen Abfall auf einer der Fahrbahnen informiert. Als Wissenskontext wird der Ort des Mülleimers ermittelt, damit das Ziel des gesamten Bewegungsablaufs bekannt ist. Dies könnte aber auch später, in-

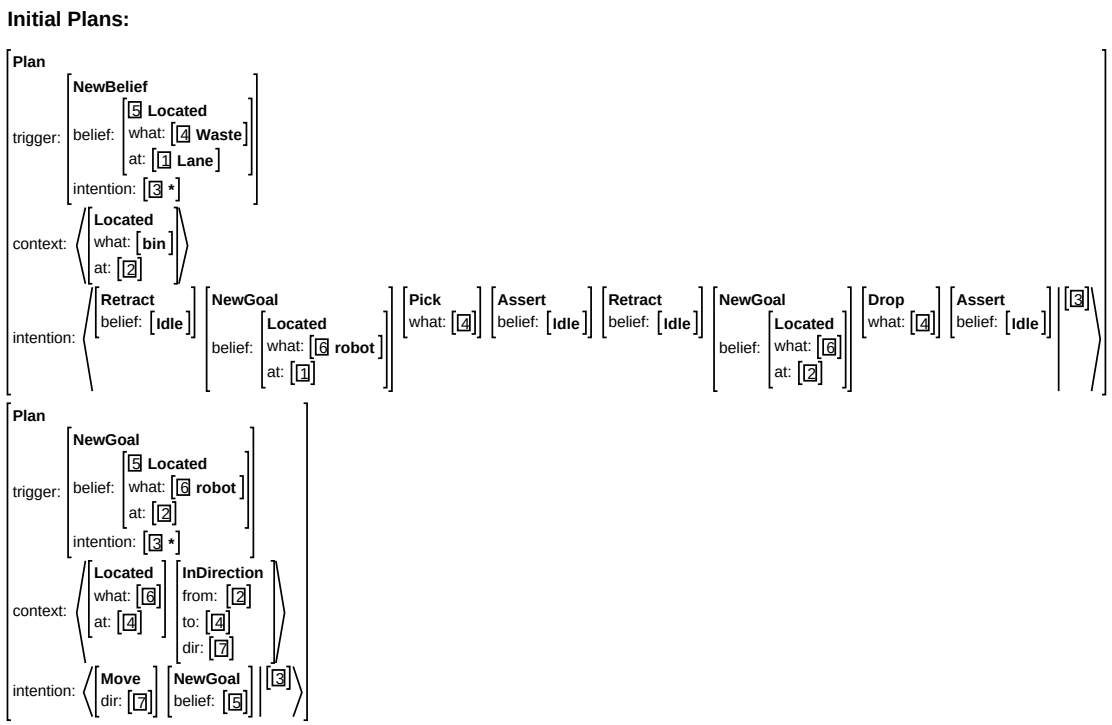

Abbildung 5.26: Die initialen Pläne des Abfallsammler-Agenten.

nerhalb der Absicht erfolgen, da auch als Schritt in einer Absicht Zugriffe auf die Wissensbasis erlaubt sind. Die aktivierte Absicht spezifiziert zwei Bewegungsphasen, die jeweils aus mehreren einzelnen Schritten bestehen. Durch $\left[\begin{array}{l}\mathsf{Retract}\\ \mathsf{belief:}\,[\mathsf{Idle}]\end{array}\right]$ und $\left[\begin{array}{l}\mathsf{Assert}\\ \mathsf{belief:}\,[\mathsf{Idle}]\end{array}\right]$ wird eine Bewegungsphase als unteilbare Handlung in bezug auf die Bewegung des Roboters realisiert. Dies ist wichtig, wenn mehrere Abfall-Ereignisse gleichzeitig zu bearbeiten sind, um eine Sequenzialisierung der Bewegungsphasen zu erzwingen. Dadurch, daß jede der beiden Bewegungsphasen einzeln „geklammert“ ist, kann sich der Agent nach der ersten Phase entscheiden, ob er die momentane Absicht abschließt und den Abfall entsorgt, oder ob er eine andere Absicht weiterverfolgt und weiteren Abfall sammelt. Jede der beiden Bewegungsphasen besteht aus dem Aufruf des Hilfsplans zum Bewegen über ein internes Ereignis und der Ausführung der entsprechenden Handlung (Abfall aufheben oder entsorgen).

Der zweite Plan stellt den Hilfsplan zum Bewegen des Agenten dar. Wenn das (interne) Ereignis eintrifft, daß der Agent eine Fahrbahn als Ziel erreichen soll, wird über den Wissenskontext mit Hilfe des Prädikate InDirection die Bewegungsrichtung ermittelt. Die aktivierte Absicht besteht aus der Bewegung um eine Fahrbahn in

die inferierte Richtung. Als zweiter Schritt der Absicht wird das Ziel wiederholt, um mehrere Fahrbahnüberquerungen zu erlauben. Ist die gewünschte Zielfahrbahn erreicht, wird der Standard-Plan (siehe oben) aktiviert, da das zu erreichende Ziel dann bereits inferierbar ist.

Wie man sieht, wird ein wesentlicher Teil eines konkreten BDI-Agenten über sein Wissen und seine Pläne spezifiziert. Die folgende Prozeßmodellierung zeigt aber, daß ein konkreter Agent auch das prozedurale Verhalten des generischen Agenten auf spezielle Bedürfnisse anpassen kann.

Prozeßmodell

Wird der generische BDI-Agent mit dem oben spezifizierten Wissen und den Plänen initialisiert, so könnte er beim Eintreffen eines Abfall-Ereignisses den entsprechenden Plan aktivieren. Bisher wurde aber noch nicht gezeigt, wie die speziellen Aktionen des Abfallsammler-Agenten ausgeführt werden.

Die Aktionen eines Agenten werden als der prozeßorientierte Aspekt seiner Modellierung betrachtet. Aktionen stellen im Gegensatz zu Wissen und Plänen Fähigkeiten des Agenten dar, die physikalischen Handlungen in der Welt entsprechen und meist in den Agenten „fest eingebaut“ sind. Aktionen werden deshalb in der Struktur des speziellen Agenten-FSNets codiert.

Der generische BDI-Agent wurde oben mit Zielkanal-Schnittstellen entworfen, um mit Plänen und Wissen initialisiert werden zu können sowie Nachrichten empfangen und Aktionen signalisieren zu können. Ein konkreter Agent nutzt diese Schnittstelle, indem er Wissen und Pläne übermittelt, Nachrichten weiterreicht und auf signalisierte Aktionen horcht. Über die synchronen Kanäle wird eine Vererbung per Delegation realisiert, da in HFSNets keine andere Art der Verhaltensvererbung möglich ist. Dies ist erstens aber in vielen Fällen auch in der Objektorientierung ein zu bevorzugender Entwurf und zweitens durch die spezielle Definition von synchronen Kanälen in HFSNets sehr einfach zu realisieren: Da ein Zielkanal alle Parameter annehmen kann, die mit seinem Kanalausdruck unifizierbar sind, können auf einfache Weise alle Synchronisationsanfragen weitergeleitet werden, die mit einem bestimmten Typ kompatibel sind. Die Delegation muß also nicht wie in vielen objektorientierten Sprachen für jede Methode einzeln spezifiziert werden.

Abbildung 5.27 zeigt das HFSNet des Abfallsammler-Agenten. Dieser erzeugt in der Stelle BDIAgent ein Exemplar des generischen BDI-Agenten-Netzes aus Abbildung 5.22 als Anfangsmarkierung. Alle Transitionen testen die Stelle, um sich über Startkanäle mit den Transitionen im `BDIAgent`-Netz mit den entsprechenden Zielkanälen zu synchronisieren.

Der Abfallsammler stellt selbst einen Zielkanal vom Typ Event an der Transition super für seine Umwelt zur Verfügung. Alle eingehenden Ereignisse werden an den generischen BDI-Agenten weitergeleitet.

Die Stellen Initial Plans und Initial Beliefs tragen die hier nicht nochmals darge-

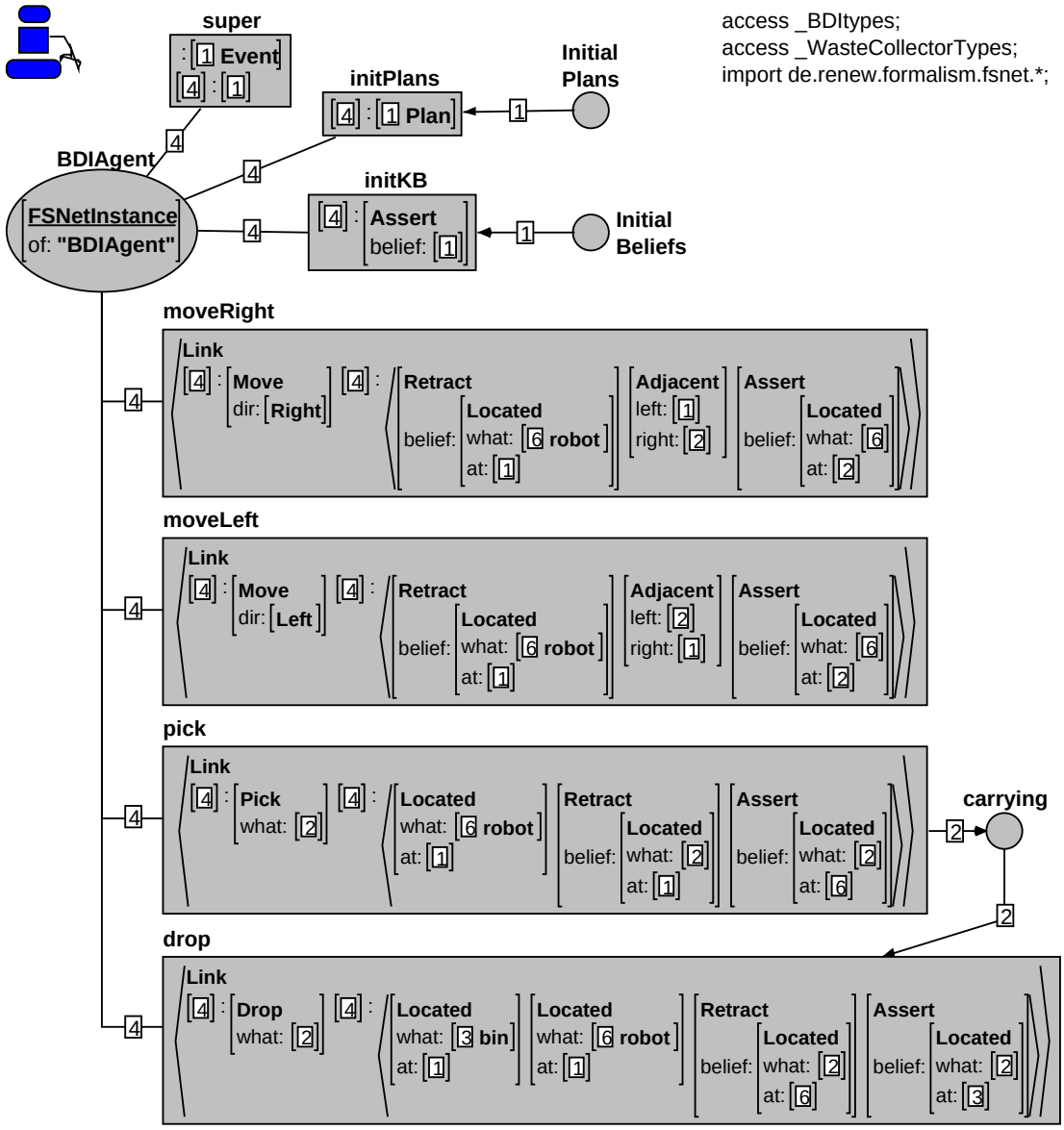

Abbildung 5.27: Ein konkreter BDI-Agent, der Abfall sammeln kann, als HFSNet *Waste-Collector*.

stellten Anfangsmarkierungen aus den Abbildungen 5.26 und 5.25. Die Transitionen initPlans und initKB übermitteln diese Marken als Pläne bzw. Wissen an den generischen BDI-Agenten.

Die Transitionen moveRight, moveLeft, pick und drop stellen das eigentliche spezielle Prozeßmodell des Abfallsammlers dar. Jede Transition trägt eine Liste von

Startkanälen, die, wie schon bei dem HFSNet zur Modellierung der Wissensbasis dargestellt, eine bevorzugte Auswertungsreihenfolge der Startkanäle vorgeben. Der jeweils erste Startkanal in den Listen stellt das Muster für die auszuführende Aktion dar. Der zweite Startkanal, der nur überprüft wird, sobald die erste Synchronisation stattfinden kann, spezifiziert die Änderungen an der Wissensbasis, die vorgenommen werden sollen, wenn die entsprechende Aktion ausgeführt wird.

Die Ausführung einer Aktion sei am Beispiel der Transition pick genauer beschrieben. Das erste Element in der Startkanal-Liste gibt das Muster für die Aktion an, in diesem Fall is der Aktionstyp Pick und das Objekt der Aktion (Feature what) wird durch die Knotenreferenz [2] benannt. Wenn diese Synchronisation zustandekommt, sollen die im zweiten Startkanal angegebenen Modifikation an der Wissensbasis vorgenommen werden. Dazu wird eine Liste mit Wissenbasis-Kommandos (impliziter Typ KBCmd) an den generischen Agenten ([4]) geschickt, der diese an seine Wissensbasis weiterreicht. Die Elemente der Liste werden durch die Wissensbasis interpretiert. Das erste Element enthält einen Retract-Befehl, der die Position des Roboters aus der Wissensbasis löscht, aber gleichzeitig noch abfragt und mit der Knotenreferenz [1] benennt. Sodann wird durch das Adjacent-Prädikat abgefragt, welche Fahrbahn sich links von der Roboterposition [1] befindet. Diese Fahrbahn wird als neue Position des Roboters der Wissensbasis durch ein Assert-Kommando hinzugefügt.

Diese Modellierung der Aktionen vereinfacht einen Aspekt der agentenorientierten Programmierung ([Shoham 1997]): Aktionen sollen Aktuatoren anstoßen, während sich das Wissen des Agenten erst durch die Information modifiziert wird, die er über seine Sensoren erhält. Dies hätte in dem vorliegenden Beispiel allerdings zu einem wesentlich komplexerem Modell der Umgebung geführt, so daß hier die Manipulation der Wissensbasis direkt in den Aktionen vorgenommen wird. Der konkrete Agent simuliert hier die Umgebung für den generischen Agenten.

In diesem einfachen Beispiel für einen konkreten BDI-Agenten ist die Stelle carrying zwischen den Transitionen pick und drop die einzige kausale Einschränkung der Aktionen. Hier wird der aufgehobene Abfall (Knotenreferenz [2]) abgelegt, bis dieser im Mülleimer entsorgt wird. In einem weitergehenden Modell, das in dieser Arbeit aus Platzgründen nicht vorgestellt wird, werden auch die Bewegungen durch ein entsprechendes Netz modelliert, so daß die Aktionen des Abfallsammlers in der Simulation graphisch verfolgt werden können.

Um eine Simulation mit dem Abfallsammler-Agenten durchführen zu können, muß dieser nun nur noch in einer „Welt“ erzeugt werden, die Abfall-Ereignisse erzeugt. Abbildung 5.28 zeigt eine sehr einfache Simulationsumgebung, die zwei Abfallereignisse zu nichtdeterministischen Zeitpunkten erzeugt. Im oben erwähnten erweiterten Modell können Abfall-Ereignisse über eine GUI, wie sie in Abschnitt 2.2.1 in Abbildung 2.9 als Beispiel herangezogen wurde, zur Laufzeit interaktiv ausgelöst werden.

Sobald ein Abfall-Ereignis erzeugt wird, wird dieses an den Abfallsammler-

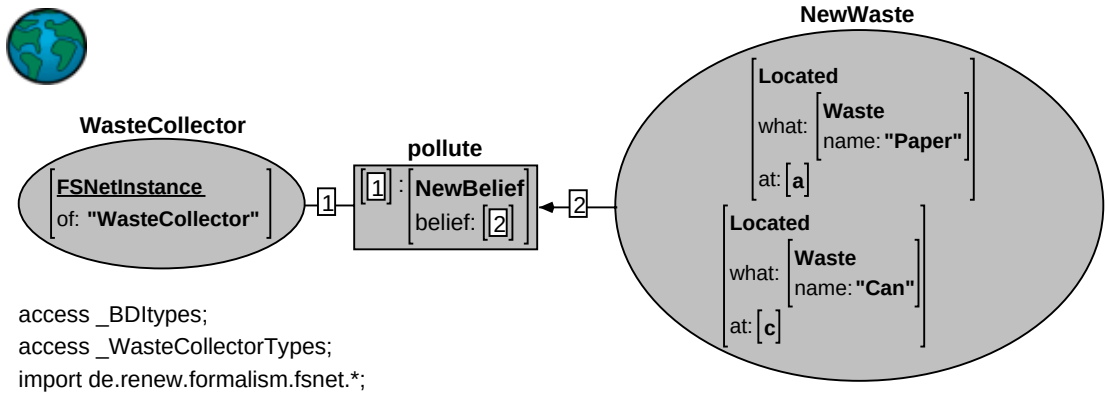

Abbildung 5.28: Eine Welt mit einem Abfallsammler und zwei Abfall-Ereignissen.

Agenten (Netz WasteCollector) und von diesem an den parametrisierten generischen BDI-Agenten (Netz BDIAgent) weitergeleitet. Der generische BDI-Agent aktiviert für jedes Ereignis eine Absicht aus dem vom Abfallsammler spezifizierten Plan. Abbildung 5.29 zeigt die beiden durch die in Abbildung 5.28 erzeugten Ereignisse in der Stelle Intentions im Netz BDIAgent generierten Absichten. Durch Protokollieren der Schaltfolgen der Transitionen im Abfallsammler-Netz kann beobachtet werden, daß der Agent tatsächlich eine korrekte Abfolge von Aktionen generiert, um die Abfälle einzusammeln. Im erweiterten Modell kann am Bildschirm beobachtet werden, wie der Agent sich bewegt, Abfall aufhebt und im Mülleimer entsorgt.

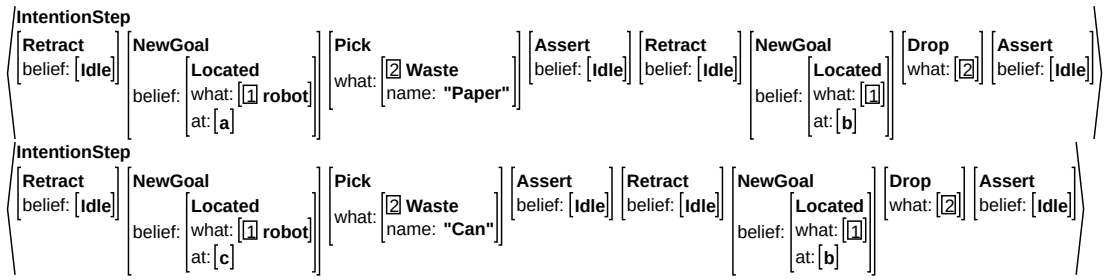

Abbildung 5.29: Die beiden Intentionen, die durch die in Abbildung 5.28 erzeugten Ereignisse in der Stelle Intentions im Netz BDIAgent generiert werden.

Das Beispiel mußte in vielen Punkten sehr einfach gehalten werden, um hier vollständig vorgestellt werden zu können. Es sind diverse Erweiterungen denkbar, die über die oben genannte Erweiterung durch eine GUI und eine graphische Simulation hinausgehen.

Das Beispiel zeigt einen einzelnen Agenten, der in einer Welt auf externe Ereignisse reagiert. Es ist offensichtlich, daß diese Ereignisse nicht nur automatisch oder von einem menschlichen Benutzer, sondern auch von anderen Agenten ausgelöst werden können. Durch die einfache Möglichkeit, mehrere Exemplare eines Netzes zu erzeugen, sind also Multi-Agenten-Systeme (siehe Abschnitt 5.3.1, [Castelfranchi und Müller 1993], insbesondere in bezug auf Petrinetze [Moldt und Wienberg 1997, Rölke 1999]) mit HFSNets realisierbar.

Multi-Agenten-Systeme stellen höhere Ansprüche an die Kommunikation zwischen Agenten. Da Feature Structures ursprünglich zur Modellierung natürlicher Sprache entwickelt wurden (siehe Abschnitt 2.3.1), können sie hier ihre Stärken besonders gut ausspielen. Der in [Rölke 1999] beschriebene Ansatz, KQML und FIPA-Agenten mit Petrinetzen zu modellieren, gewinnt damit noch mehr an Attraktivität.

In dieser Dissertation wurden Grundlagen von FSNets erarbeitet und erste Beispiele zur Modellierung mit dieser neuen Technik beschrieben. Eine Modellierung und Ausführung komplexer Agentensysteme geht damit über den hier gesteckten Rahmen hinaus, scheint aber mit den beschriebenen Voraussetzungen ein gangbarer Weg.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Diese Arbeit wurde unter der Zielsetzung erstellt, eine *Modellierungstechnik für verteilte Systeme* bereitzustellen. Unter „Bereitstellung“ wird hier im einzelnen die Motivation, Definition, formale Untersuchung und der exemplarische Einsatz der Modellierungstechnik verstanden, wozu wiederum ein entsprechendes Werkzeug benötigt wird. In Abschnitt 1.2 der Einleitung werden diese Aspekte nach [Moldt 1996] als *Facetten* eines Ansatzes zur Systemspezifikation bezeichnet, wobei in dieser Arbeit vor allem *Techniken*, *Werkzeuge* und *Anwendungen* behandelt werden.

Als Überblick wird die Arbeit zunächst zusammengefaßt. Sodann wird auf die theoretischen und praktischen Ergebnisse der Arbeit im einzelnen eingegangen.

Kapitel 2 und 3 betrachten verschiedene existierende informations- bzw. prozeßorientierte Modellierungs*techniken*. Damit werden die Techniken, welche die Basis für die vorgeschlagene Modellierungstechnik darstellen, in einen Kontext gestellt, und ihre Auswahl in bezug auf das Anwendungsgebiet verteilter Systeme wird motiviert.

Für die Informationsmodellierung werden in Kapitel 2 *typisierte Feature Structures* ausgewählt, welche die in der Einleitung in Abschnitt 1.1.1 motivierte Operation der *Unifikation* unterstützen. Es wird ein starker Bezug von Feature Structures zum heute dominierenden Paradigma der *Objektorientierung* und konkret zur standardisierten objektorientierten Modellierungssprache UML aufgezeigt. Feature Structures sind damit in der Lage, eine abstrakte *Objektsicht* zu spezifizieren, mit der die Manipulation von Information und die Anwendung von Regeln dargestellt werden kann.

Zur Prozeßmodellierung werden in Kapitel 3 Petrinetze ausgewählt. Diese zeigen ihre Stärken bei der Modellierung von *Kausalität*, *Nebenläufigkeit* und *Synchronisation*. Unter anderem lassen sich, wie in der Einleitung in Abschnitt 1.1.2 gezeigt, verschiedene Kommunikationsarten (Interaktionsmuster) in verteilten Systemen mit Petrinetzen darstellen. Weitere wichtige Kriterien für die Auswahl sind die *formale*

Definition und die *Ausführbarkeit* (operationale Semantik) von Petrinetzen.

Feature Structures und Petrinetze ergänzen sich als Techniken, welche die informations- bzw. prozeßorientierten Aspekte verteilter Systeme modellieren können. Die in dieser Arbeit in Kapitel 4 vorgeschlagenen *Feature-Structure-Netze* oder kurz *FSNets* verwenden Feature Structures als Anschriften und Marken in Petrinetzen. Damit folgen sie den Ansätzen verschiedener höherer Petrinetze, die Daten und Information in Prozessen durch individuelle, typisierte Marken und die Verarbeitung dieser Daten als Transitionsanschriften repräsentieren. Großer Wert gelegt wird hier auf die *graphische Notation* der FSNets.

Sowohl Feature Structures als auch Petrinetze stellen formal definierte und untersuchte Techniken dar. Dies erleichtert die *formale Definition* von FSNets in Kapitel 4 erheblich. Als Voraussetzung sind vor allem bei den Feature Structures in Abschnitt 2.3 zusätzliche formale Untersuchungen nötig, um diese in FSNets einsetzen zu können (siehe unten). Im Bereich der Petrinetze läßt sich dagegen auf die bestehenden Definitionen zurückgreifen.

Mithilfe der formalen Definition von *Basis-FSNets* und vor allem durch die Einschränkung auf *elementare FSNets* (EFSNets) werden als Beitrag der Arbeit zur Petrinetztheorie verschiedene Phänomene der Petrinetze und der verteilten Systeme formal untersucht (siehe unten).

Am Ende der Kapitel 2 und 4 wird auf die Implementierung von Feature Structures bzw. FSNets eingegangen. Damit wird ein Werkzeug zur Bearbeitung und Ausführung von FSNets zur Verfügung gestellt.

Die Arbeit schließt mit dem praktischen Einsatz von FSNets in verschiedenen Anwendungsgebieten der verteilten Systeme. Hier wird anhand von Beispielen die Modellierung und Ausführung von Geschäftsprozessen, elektronischen Verträgen und Agenten betrachtet.

Ergebnisse

Mit FSNets wird eine neue Familie von höheren Petrinetzen vorgestellt, die sich besonders zur Modellierung *verteilter* Systeme eignen, da typische Phänomene in verteilten Systemen in den Entwurf der Modellierungstechnik eingegangen sind. Während Petrinetze durch die Darstellung von Kausalität und Nebenläufigkeit verschiedenste Kommunikationsarten verteilter Systeme modellieren können, erlauben Feature Structures die Konsistenzprüfung und Zusammenführung verteilter replizierter Daten durch Unifikation.

Die eigenständigen Ergebnisse dieser Arbeit bestehen in der theoretischen und praktischen Untersuchung von FSNets.

Theoretische Ergebnisse

Die theoretischen Grundlagen dienen der genauen Darstellung der Semantik von FSNets. Das allgemeine Phänomen der Konsistenz von Daten in verteilten Systemen und deren Zusammenführung wird durch Feature Structures und Unifikation formalisiert. Die inhärente Nebenläufigkeit verteilter Systeme wird durch Petrinetze formal beschrieben. Durch FSNets können beide Phänomene kombiniert dargestellt werden, was neue Möglichkeiten zur deren Untersuchung eröffnet.

Die Beiträge und Ergebnisse der Arbeit im Bereich der Theorie sind im einzelnen:

- die Anpassung und der Bezug formaler Definitionen von Typsystemen und Feature Structures auf den Bereich der Objektorientierung, insbesondere die Herleitung einer hier geeigneten *Typinferenz*,
- die Definition von *Schaltfolgen-* und *Schrittsemantik* für Basis-FSNets,
- die Untersuchung von *Wert- und Referenzsemantiken* anhand von EFSNets, welche eine formale Definition von *Partitionen*, also Gruppen lokaler Stellen, erlaubt,
- die Definition von Prozessen von EFSNets, die zu *gefärbten Prozessen* führt,
- die Spezifikation eines *universellen EFSNets*, das analog zur universellen Turingmaschine beliebige andere EFSNets simulieren kann, womit die Semantik eines EFSNets „in sich selbst“ beschrieben wird, und schließlich
- eine Darstellung von *Netzen in Netzen* durch FSNets beziehungsweise Feature Structures, die gefärbten elementaren Objektsystemen beziehungsweise Referenznetzen ähnelt und damit eine neue Semantik für Netze in Netzen zur Verfügung stellt.

Praktische Ergebnisse

Das zentrale praktische Ergebnis der Arbeit besteht in der Bereitstellung von HFS-Nets als Modellierungstechnik. Dazu gehört die Entwicklung einer graphischen Notation, welche die informations- und prozeßorientierten Aspekte abdeckt. Feature Structures eignen sich gut zur formalen und gleichzeitig abstrakten Beschreibung von Objektgraphen. Die klassische Notation als AVM oder als Graph ist für einen Modellierer, der objektorientierte Spezifikation kennt, eher ungeeignet. Als ein wichtiger Beitrag dieser Arbeit wird der starke Bezug zwischen der Objektorientierung und hierarchischen Typsystemen sowie Feature Structures aufgezeigt. Dadurch kann ein Ausschnitt von UML angegeben werden, der sich zur Notation von Konzeptsystemen, Feature Structures und konkreten Objektgraphen eignet. Diese Notation kann insbesondere in höheren Petrinetzen eingesetzt werden und stellt somit eine UML-basierte Informationsmodellierung für Petrinetze zur Verfügung.

Um FSNets in der Praxis verwenden zu können, werden einige in dieser Arbeit nicht formal definierte Erweiterungen eingeführt. Dies sind zusätzliche Kantenarten wie Test-, Reservierungs und Inhibitorkanten und die Konzepte der Referenznetze, die zum erweiterten Modell der *höheren FSNets* führen.

Der praktische Einsatz von FSNets zur Modellierung verteilter Systeme wird an mehreren Fallbeispielen demonstriert. Dabei wird durch Beispiele aus den Bereichen Geschäftsprozeßmodellierung, E-Commerce und Agenten die große Bandbreite der möglichen Einsatzgebiete von FSNets gezeigt.

Bei der Geschäftsprozeßmodellierung zeigen sich neben den bekannten Vorteilen von Petrinetzen zur Darstellung von verteilten und nebenläufigen Prozessen die Vorteile der intuitiven, aber dennoch ausdrucksstarken Datenmodellierung durch Feature Structures. Rollen, Aufgaben und Geschäftsobjekte (Dokumente, Formulare, Ressourcen etc.) können auf einfache Weise modelliert und in Transitionsregeln verarbeitet werden. Die Kombination mit Konzepten der Referenznetze erweist sich für die Darstellung von Workflow-Exemplaren und die Verteilung eines Workflows auf verschiedene Unternehmen als vorteilhaft.

Beim Einsatz von Referenznetzen und FSNets zur Ausführung von elektronischen Verträgen in elektronischen Dienstemärkten (E-Commerce) wird demonstriert, wie andere Repräsentation von Workflows in Petrinetze und insbesondere FSNets überführt werden können. An dieser Fallstudie ist hervorzuheben, daß sie innerhalb des EU-Projekts Cosmos in einem praxisorientierten Kontext entwickelt wurde. Konkret wird eine Darstellung, die auf UML-Aktivitätsdiagrammen basiert, in ein FSNet übersetzt und als Workflow ausgeführt.

Im dritten Anwendungsgebiet wird eine generische Architektur für intelligente Agenten nach dem BDI-Ansatz (*belief, desire, intention*) als FSNet modelliert, und anhand eines Beispiels wird gezeigt, wie dieses Modell bei der Realisierung eines konkreten Agenten wiederverwendet werden kann. Softwareagenten bieten durch ihre Mobilität und Autonomie die idealen Voraussetzungen, um in verteilten System eingesetzt zu werden. Sie stellen durch ihr intelligentes, regelbasiertes Verhalten einerseits und ihre Einbettung in eine ereignisbasierte Echtzeitumgebung andererseits hohe Anforderungen an Implementierungs- und Modellierungstechniken. In FSNets kann das wissens- und regelbasierte Verhalten durch das Informationsmodell (Typen und Feature Structures) und die nebenläufige Verarbeitung von Nachrichten und Aktivierung sowie Ausführung von Plänen durch das Prozeßmodell (Petrinetze) erfolgreich umgesetzt werden.

Die eingehende Modellierung und Untersuchung praktischer Anwendungsbeispiele war nur durch die im Rahmen dieser Arbeit entstandene *Werkzeugunterstützung* möglich. Der zusammen mit Kummer entwickelte Petrinetz-Editor und -Simulator Renew ("Reference Net Workshop") wurde vom Autoren um einen Modus (*mode*) erweitert, mit dem HFSNets erstellt und ausgeführt werden können. Dafür mußte als Voraussetzung die in Abschnitt 2.4 beschriebene Feature-Structure-Klassenbibliothek erstellt werden. Weiterhin wurde die graphische Oberfläche um die

Bearbeitung und Darstellung von Typsystemen und Feature Structures sowie viele weitere Details erweitert sowie der Netzcompiler für HFSNets erstellt. Als Ergebnis steht mit FSRenew ein Werkzeug zur Verfügung, das kaum noch als Prototyp bezeichnet werden kann. Sämtliche Abbildungen dieser Arbeit wurden mit FSRenew erstellt, und alle gezeigten FSNets sind mit FSRenew ausführbar. FSRenew wird mit der nächsten Version von Renew über die Web-Site `http://www.renew.de` als Open-Source zur allgemeinen Verfügung gestellt.

6.2 Ausblick

Mit der Modellierung verteilter Systeme wird in dieser Arbeit ein sehr weiträumiges Thema behandelt. Aus den hier präsentierten theoretischen und praktischen Grundlagen ergeben sich viele interessante Fragen und Problemstellungen, die hier nicht erschöpfend behandelt werden können. Als Abschluß der Arbeit werden einige dieser offene Fragen als Ausgangspunkte weitergehender Forschung genannt.

Im theoretischen Teil der Arbeit wurden Basis-FSNets und EFSNets dargestellt und untersucht. Im Anschluß an die hier erzielten Grundlagen kann die Semantik dieser Netzformalismen weiter untersucht werden. So stellt sich die Frage nach der Effizienz der dargestellten Algorithmen und der Entscheidbarkeit typischer Petrinetz-Eigenschaften, wie sie zum Teil für EFSNets behandelt wurden. Zu vermuten ist, daß für FSNets wie für die meisten höheren Petrinetze alle interessanten Probleme wie die Erreichbarkeit bestimmter Markierungen oder die Berechnung von Invarianten unentscheidbar sind. Um in diesem Bereich dennoch Ergebnisse zu erzielen, könnte nach weiteren Einschränkungen, beispielsweise in bezug auf Typsysteme oder Feature Structures, gesucht werden, die zur Entscheidbarkeit solcher Probleme führen.

Für elementare FSNets und Basis-FSNets werden in dieser Arbeit formale Definitionen gegeben und Eigenschaften gezeigt. Darauf aufbauend kann nun eine formale Definition der in Abschnitt 4.4 eingeführten Erweiterungen auf HFSNets hergeleitet werden. Bei den Erweiterungen gegenüber Basis-FSNets handelt es sich im einzelnen um die zusätzlichen Kantenarten, synchrone Kanäle und Netzexemplare. Für diese Erweiterungen gibt es in der Petrinetzliteratur bereits Formalisierungen[1], die als Basis für die entsprechenden erweiterten Definition für HFSNets herangezogen werden können. Weiterhin wurde die Darstellung von Prozessen von Netzen in Netzen als Feature Structures in Abschnitt 4.3.2 zwar in allen wesentlichen Details, insgesamt aber semi-formal behandelt. Die letzten technischen Details dieser Konstruktion sind noch auszuarbeiten.

Als eine der beiden Grundlagen von FSNets sind auch im Bereich der Feature Structures weitere Untersuchungen und Verallgemeinerungen denkbar. [Carpenter 1992] stellt extensionale Typen, Negation durch Spezifikation der Un-

[1] Für Referenznetze werden diese erst mit dem Erscheinen der Dissertation von Kummer ([Kummer 2001]) allgemein zugänglich.

gleichheit (Nicht-Identität) von Knoten und andere Erweiterungen vor, welche direkt in FSNets eingesetzt werden könnten. Die formalen Definition von Carpenter können leicht an die in dieser Arbeit gewählten Varianten angepaßt werden. Es steht nur die Implementation in der Feature-Structure-Bibliothek für FSRenew aus, um Theorie und Werkzeugunterstützung konsistent parallel weiterzuentwickeln.

Eine Idee zur Verallgemeinerung von Carpenters Form der Negation ist aus Diskussionen mit Kummer entstanden. Während in Carpenters Ansatz nur ausgedrückt werden kann, daß zwei Knoten identisch sind (Knotenreferenzen), nie identisch werden können (Negation) oder keine Information über ihre Identität vorliegt, schlagen wir vor, daß Knoten als *einander subsumierend* spezifiziert werden können. Wenn zwei Knoten in dieser Subsumptionsrelation stehen, bedeutet dies, daß die Unterstruktur ab dem einen Knoten stets spezieller als die Unterstruktur ab dem anderen Knoten sein soll. Eine gegenseitige Subsumption entspricht der klassischen Knotenreferenz, so daß eine echte Verallgemeinerung entsteht. Vor allem bei zyklischen Strukturen müssen die Konsequenzen dieses Ansatzes noch untersucht werden.

Stehen Feature Structures mit Subsumption zwischen Knoten zur Verfügung, könnten diese elegant zur Definition von Wächtern in FSNets (siehe unten) eingesetzt werden. Da dieses Vorgehen als Erweiterung der FSNets geplant ist, wurde in dieser Arbeit auf eine Definition von FSNets mit expliziten Wächtern verzichtet.

Ein vollkommen anderer Bereich von Erweiterungsmöglichkeiten sind konkrete Verbesserungen des Werkzeugs FSRenew.

Bei der Implementierung der Feature-Structure-Bibliothek und des Netzcompilers wurden klare Strukturen und Bereitstellung der benötigten Funktionalität gegenüber höchster Effizienz in den Vordergrund gestellt. Insofern bietet der Bereich der Algorithmen (Unifikation, Subsumption) sicherlich noch Potential für Optimierungen. Die Integration von FSNets in Renew wurde so vorgenommen, daß möglichst viele Teile von Renew im Sinne der Objektorientierung wiederverwendet werden konnten. Deshalb ist beispielsweise die Simulation von Netzen nicht speziell für FSNets optimiert worden, was vor allem im Bereich der synchronen Kanäle möglich wäre.

Die in Abschnitt 2.4.2 vorgestellte Integration von Typen und Java-Objekten analysiert bestehende Java-Klassen durch Reflektion auf vorhandene Attribute. Ein Konzept aus Java, das Attribute explizit modelliert, sind die sogenannten *Java Beans*. Beans ließen sich aufgrund ihrer abstrakteren Modellierung sauberer in das Typsystem integrieren als beliebige Java-Objekte, die sich nicht immer an Konventionen halten, die in der vorliegenden Integration vorausgesetzt werden.

Auch in der Benutzungsoberfläche von FSRenew sind noch zahlreiche Verbesserungen möglich. Beispielsweise können Feature Structures trotz der Möglichkeit des Öffnens und Schließens von Unterstrukturen (siehe Abschnitt 4.5) sehr groß und unübersichtlich werden. Hier wäre ein Mechanismus hilfreich, der auf Wunsch Unterstrukturen in einem separaten Fenster darstellt.

Die letzte hier angeregte werkzeugbezogene Erweiterung ist die Bereitstellung

von Java-Klassenbibliotheken für bestimmte, wiederkehrende Aufgaben, um ein schnelles Prototyping von Anwendungen zu erlauben. Einige solcher Klassen wurden bereits erstellt, um auf einfache Weise GUIs erstellen und verwenden zu können (siehe Abschnitt 5.3.4). Der wesentliche Grund dafür, daß hier spezielle (Sub-)Klassen implementiert werden mußten, ist, daß sich die Standard-Java-Klassenbibliotheken nicht immer an die in Abschnitt 2.4 erwähnten Entwurfsmuster halten. Andere Klassen sind denkbar, um durch Feature Structures Zugriff auf weitere Java-Standardfunktionen zu erhalten.

Neben den weiteren theoretischen Untersuchungen und den konkreten Verbesserungen am Werkzeug kann auch die Modellierungstechnik selbst noch auf verschiedene Arten erweitert werden.

In den Anwendungsbeispielen, vor allem bei der Geschäftsprozeßmodellierung in Abschnitt 5.1 fiel auf, daß bei dem Abgleich von Daten und Mustern (z.B. Personen und Rollen) oft nicht Unifizierbarkeit, sondern Subsumption getestet werden soll. Es liegt deshalb nahe *Subsumptionsausdrücke* als Transitionsanschriften einzuführen. Die Transition könnte dann, analog zu Wächtern in gefärbten Petrinetzen, nur schalten, wenn die angegebene Subsumption unter der gewählten Bindung erfüllt ist. Eine entsprechende Schaltregel kann formal definiert werden, wofür sich die oben erwähnte Subsumption innerhalb von Feature Structures anbietet.

Eine Möglichkeit, die Annäherung von FSNets an UML voranzutreiben, stellt die Notation der Petrinetzstruktur eines FSNet als Aktivitätsdiagramm dar. Wie sich Petrinetze aus Aktivitätsdiagrammen ableiten lassen, wurde in Abschnitt 5.2.3 anhand eines Beispiels gezeigt. Ein Problem stellen die ausdrucksstarken Anschriften von Netzelementen in FSNets dar, die auf Aktivitätsdiagramme übertragen werden müßten. Die in UML definierten Objektflüsse (siehe Abschnitt 3.1.3) sind ein Schritt in diese Richtung, sind aber nicht ausdrucksstark genug.

Vor allem beim Einsatz von FSNets zur Workflow-Modellierung bietet es sich an, bestehende Erkenntnisse aus der Welt der Petrinetze auf FSNets zu übertragen. Da für FSNets das prinzipielle Schaltverhalten von Petrinetzen übernommen wurde (beispielsweise im Gegensatz zu SGML- und XML-Netzen, siehe Abschnitt 3.4.5), kann eine Analyse der kausalen Struktur eines FSNets begrenzten Aufschluß über Modellierungsfehler geben. Dazu kann beispielsweise das Werkzeug WOFLAN ([Verbeek et al. 1999]) verwendet werden. Auch andere Verfahren aus dem Bereich der Workflow-Netze ([Aalst 1997]) wie die Vererbung von Verhalten ([Aalst und Anyanwu 1999]) können möglicherweise in bezug auf FSNets erweitert werden. Während van der Aalst stets S/T-Netze betrachtet, kann durch die in FSNets unifizierbaren Marken der Ansatz möglicherweise auf Netze mit individuellen Marken erweitert werden, analog zu den in Abschnitt 4.3 gezeigten Erweiterungen für Netze in Netzen.

Abschließend sei auf ein weiteres großes Gebiet hingewiesen, in dem noch wenige Ergebnisse erzielt wurden, aber konkreter Bedarf an tiefergehenden Lösungsansätzen besteht. Wie in der Einleitung bereits abgegrenzt, beschäftigt sich diese Arbeit mit

einer Modellierungs*technik* für verteilte Systeme. Nach der Spezifikation einer Technik stellt sich hier als nächster Schritt die Frage nach dem allgemeinen *Vorgehen* bei der Modellierung verteilter Systeme.

Das hier anhand von Anwendungsbeispielen demonstrierte Vorgehen kann als Ausgangspunkt für eine allgemeines Vorgehensmodell herangezogen werden. Vor allem das Beispiel aus Abschnitt 5.3 zeigt, daß FSNets die Modellierung intelligenter Software-Agenten in verteilten Systemen auf eine noch nicht dagewesene Art und Weise erlauben. Das Paradigma der Agenten stellt einen wahrscheinlichen Nachfolger der Objektorientierung dar. FSNets bieten alle Voraussetzungen, um die zur Modellierung von Agenten nötigen Techniken wie deliberatives und reaktives Verhalten, Prozeßsteuerung durch dynamische Protokolle und Migration in einem integrierten Modell zu beschreiben. Zusammen mit einem entsprechenden Vorgehensmodell kann nun eine Software-Spezifikationsumgebung entstehen, die für die Modellierung und Realisierung verteilter Systeme vollkommen neue Perspektiven eröffnet.

Anhang A

Grundlegende Definitionen

Definition A.1 [Urbildbereich einer Funktion]
Sei f : F $\hookrightarrow$ I eine partielle Funktion, dann heißt die Menge D := $\{r \in \mathrm{F} \mid \mathrm{f}(r)$ existiert$\}$ der *Urbildbereich (domain)* von f. dom sei die Funktionen, die eine partielle Funktion auf ihren Urbildbereich abbildet, also dom(f) = D. $\Diamond$

Definition A.2 [Funktionenvereinigung]
Die *Funktionenvereinigung* zweier partieller Funktionen f : F $\hookrightarrow$ I und g : G $\hookrightarrow$ J sei, falls $\forall x \in \mathrm{F} \cap \mathrm{G} : \mathrm{f}(x) = \mathrm{g}(x)$, definiert als die partielle Funktion f$\cup$g : F$\cup$G $\hookrightarrow$ I$\cup$J, so daß für alle $x \in \mathrm{F} \cup \mathrm{G}$:

$$\mathrm{f} \cup \mathrm{g}(x) := \begin{cases} f(x) & \text{falls } x \in \mathrm{dom(f)} \\ g(x) & \text{falls } x \in \mathrm{dom(g)} \\ \text{divergent} & \text{sonst} \end{cases}$$

$\Diamond$

f $\cup$ g ist wegen der Bedingung $\forall x \in \mathrm{F} \cap \mathrm{G} : \mathrm{f}(x) = \mathrm{g}(x)$ eindeutig definiert.

Divergente Funktionswerte werden in dieser Arbeit auch als „undefiniert“ bezeichnet. Wenn auch „divergent“ für Funktionswerte, die als nicht-existierend definiert sind, die exakte Bezeichnung darstellt, wird „undefiniert“ als Gegenteil von „definiert“ bevorzugt.

Definition A.3 [Partielle Ordnung]
Eine *partielle Ordnung* $\preceq$ über einer Menge A ist eine reflexive, transitive und antisymmetrische Relation $(\preceq) \subseteq A \times A$. $\Diamond$

Relationen werden auch in Infixnotation notiert: $(x, y) \in (\preceq) \iff x \preceq y$. Wie in diesem Beispiel werden Relationssymbole in runde Klammern gesetzt, wenn ihr Gebrauch sonst schlecht lesbar oder mißverständlich wäre.

Definition A.4 [Partiell geordnete Menge]
Eine *partiell geordnete Menge (partially ordered set, poset)* ist ein Paar P := $\langle \mathrm{A}, \preceq \rangle$ aus einer Menge A und einer partiellen Ordnung $\preceq$ über A. $\Diamond$

Definition A.5 [Obere Schranken]
Sei $P = \langle A, \preceq \rangle$ eine partiell geordnete Menge. Die Menge der *oberen Schranken* einer Teilmenge T von A bezüglich P sei durch die Funktion supset_P mit $\text{supset}_P : \mathcal{P}(A) \to \mathcal{P}(A)$ definiert als $\text{supset}_P(T) := \{s \in A \mid \forall t \in T : t \preceq s\}$. ◊

Definition A.6 [Konsistente Menge]
Sei $P = \langle A, \preceq \rangle$ eine partiell geordnete Menge. Eine Menge $T \subseteq A$ heißt *konsistent bezüglich* P, wenn sie bezüglich $\preceq$ mindestens eine obere Schranke in A besitzt, also wenn gilt: $\text{supset}_P(T) \neq \emptyset$. ◊

Definition A.7 [Maximum]
Sei $P = \langle A, \preceq \rangle$ eine partiell geordnete Menge. Das *Maximum* einer Menge $T \subseteq A$ bezüglich P sei die Menge der bezüglich $\preceq$ größten Elemente in T, definiert als die partielle Funktion $\max_P : \mathcal{P}(A) \hookrightarrow \mathcal{P}(A)$ mit

$$\max_P(T) := \{t \in T \mid \forall t' \in T, t' \neq t : (t, t') \notin (\preceq)\}$$ ◊

Definition A.8 [Minimum]
Sei $P = \langle A, \preceq \rangle$ eine partiell geordnete Menge. Das *Minimum* einer Menge $T \subseteq A$ bezüglich P sei die Menge der bezüglich $\preceq$ kleinsten Elemente in T, definiert als die partielle Funktion $\min_P : \mathcal{P}(A) \hookrightarrow \mathcal{P}(A)$ mit

$$\min_P(T) := \{t \in T \mid \forall t' \in T, t' \neq t : (t', t) \notin (\preceq)\}$$

◊

Definition A.9 [Supremum]
Sei $P = \langle A, \preceq \rangle$ eine partiell geordnete Menge. Das *Supremum* einer Menge $T \subseteq A$ bezüglich P sei definiert als das einzige Element des Minimums der oberen Schranken von T, falls ein solches existiert. Wir notieren $\sup_P : \mathcal{P}(A) \hookrightarrow A$ als eine partielle Funktion mit $\forall T \subseteq A : \sup_P(T) := m$, falls $\min_P(\text{supset}_P(T))$ die einelementige Menge $\{m\}$ ist, divergent sonst. ◊

Definition A.10 [Abgeschlossene partielle Ordnung]
Sei $P = \langle A, \preceq \rangle$ eine partiell geordnete Menge. Die partielle Ordnung $\preceq$ heißt *abgeschlossen*, wenn jede konsistente Teilmenge T von A ein Supremum bezüglich P besitzt, also wenn gilt: $\forall T \subseteq A : \text{supset}_P(T) \neq \emptyset \Rightarrow \sup_P(T)$ ist definiert. ◊

Definition A.11 [Multimenge]
Sei $\mathbb{N}$ die Menge der Natürlichen Zahlen inklusive 0. Eine *Multimenge* über einer Basismenge A ist eine Funktion $M_A : A \to \mathbb{N}$. ◊

Wir notieren eine endliche Multimenge als eine Aufzählung der Elemente ihrer Basismenge, wobei die durch die Multimenge zugeordnete Multiplizität für jedes Element mit einem Apostroph als Trennzeichen vorangestellt wird. Die Aufzählung wird in Mengenklammern mit einem Index M eingeschlossen. Sei $A = \{a_1, a_2, \ldots, a_n\}$, so

wird eine Multimenge $\mathrm{M_A}$ mit $\forall i \in \{1 \ldots \mathrm{n}\} : \mathrm{M_A}(\mathrm{a}_i) = \mathrm{m}_i$ demnach notiert als $\mathrm{M_A} = \{\mathrm{m}_1{}'\mathrm{a}_1, \mathrm{m}_2{}'\mathrm{a}_2, \ldots, \mathrm{m}_\mathrm{n}{}'\mathrm{a}_\mathrm{n}\}_\mathrm{M}$. Die Multiplizität 1 kann samt Apostroph weggelassen werden. Falls die Basismenge bekannt ist, können Elemente mit der Multiplizität 0 bei der Notation der Multimenge entfallen. Auf diese Weise können auch Multimengen über unendliche Basismengen, die nur endlich vielen Elementen eine Multiplizität ungleich 0 zuweisen, durch die aufzählende Darstellung definiert werden.

Definition A.12 [Multimengen-Kardinalität]
Die *Kardinalität* einer Multimenge $\mathrm{M_A}$ über einer Basismenge A, notiert als $|\mathrm{M_A}|_\mathrm{M}$, sei definiert als $|\mathrm{M_A}|_\mathrm{M} := \sum_{a \in \mathrm{A}} \mathrm{M_A}(a)$. ◊

Definition A.13 [Multimengen-Element]
Für eine Multimenge $\mathrm{M_A}$ über einer Basismenge A sei $a \in \mathrm{A}$ genau dann ein Element von $\mathrm{M_A}$, notiert als $a \in_\mathrm{M} \mathrm{M_A}$, wenn gilt $\mathrm{M_A}(a) > 0$. ◊

Definition A.14 [Multimengen-Inklusion]
Sei A eine Menge. Eine Multimenge $\mathrm{M_A}$ über A heißt *Multimengen-Teilmenge* einer Multimenge $\mathrm{M'_A}$ über A, notiert als $\mathrm{M_A} \subseteq_\mathrm{M} \mathrm{M'_A}$, wenn für alle $a \in \mathrm{A}$ gilt: $\mathrm{M_A}(a) \leq \mathrm{M'_A}(a)$. ◊

Definition A.15 [Multimengen-Vereinigung]
Die *Multimengen-Vereinigung* einer endlichen Multimenge von Multimengen $\mathrm{MM_A} = \{\mathrm{m}_1{}'\mathrm{M}_{\mathrm{A},1}, \mathrm{m}_2{}'\mathrm{M}_{\mathrm{A},2}, \ldots, \mathrm{m}_n{}'\mathrm{M}_{\mathrm{A},n}\}_\mathrm{M}$ über einer Basismenge A, notiert als $\bigcup_\mathrm{M} \mathrm{MM_A}$, sei folgendermaßen definiert:

$$\forall a \in \mathrm{A} : (\bigcup_\mathrm{M} \mathrm{MM_A})(a) := \sum_{i \in \{1 \ldots n\}} \mathrm{m}_i \cdot \mathrm{M}_{\mathrm{A},i}(a)$$

Für zweielementige Multimengen-Vereinigungen sei auch folgende Schreibweise erlaubt:

$$\mathrm{M_A} \cup_\mathrm{M} \mathrm{M'_A} := \bigcup_\mathrm{M} \{\mathrm{M_A}, \mathrm{M'_A}\}_\mathrm{M}$$ ◊

Definition A.16 [Multimengen-Differenz]
Die *Multimengen-Differenz* zweier Multimengen $\mathrm{M_A}$ und $\mathrm{M'_A}$ über einer Basismenge A, notiert als $\mathrm{M_A} -_\mathrm{M} \mathrm{M'_A}$, sei definiert genau dann wenn gilt $\mathrm{M'_A} \subseteq_\mathrm{M} \mathrm{M_A}$, und dann folgendermaßen:

$$\forall a \in \mathrm{A} : (\mathrm{M_A} -_\mathrm{M} \mathrm{M'_A})(a) := \mathrm{M_A}(a) - \mathrm{M'_A}(a)$$ ◊

Anhang B

Definitionen zu Petrinetzen

Die folgenden Definition stammen aus [Jessen und Valk 1987] und sind nur unwesentlich in der Notation nach den Konventionen in dieser Arbeit angepaßt.

Definition B.1 [Netz]
Ein *Netz* (net) ist ein Tripel $N = \langle \mathrm{S}, \mathrm{T}, \mathrm{F} \rangle$, besteht aus einer Menge S von *Stellen*, man sagt auch *Plätzen* (places), einer dazu disjunkten Menge T von *Transitionen* (transitions) und einer *Flußrelation* (flow relation) $\mathrm{F} \subseteq (\mathrm{S} \times \mathrm{T}) \cup (\mathrm{T} \times \mathrm{S})$. ◊

Definition B.2 [Vor-, Nach- und Nebenbedingungen]
Sei $N = \langle \mathrm{S}, \mathrm{T}, \mathrm{F} \rangle$ ein Netz nach Definition B.1. Für ein Element $x \in \mathrm{S} \cup \mathrm{T}$ bezeichnet $\bullet x := \{y \mid (y, x) \in \mathrm{F}\}$ die Menge der *Eingangselemente*, sowie $x\bullet := \{y \mid (x, y) \in \mathrm{F}\}$ die Menge der *Ausgangselemente*.
Für $t \in \mathrm{T}$ heißt $\bullet t$ bzw. $t\bullet$ auch die Menge der *Vor-* bzw. *Nachbedingungen* (*pre-, post-conditions*) oder auch *Eingangs-* bzw. *Ausgangsstellen* von t.
Eine Stelle aus der Menge $\bullet t \cap t\bullet$ wird als *Nebenstelle* oder *Nebenbedingungen* (*side condition*) bezeichnet. Für $s \in \mathrm{S}$ heißt $\bullet s$ bzw. $s\bullet$ die Menge der *Eingangs-* bzw. *Ausgangstransitionen* von s.
Für eine Menge $\mathrm{A} \subseteq \mathrm{S} \cup \mathrm{T}$ definiert man entsprechend
$\bullet \mathrm{A} := \{x \mid \exists y \in \mathrm{A} : (x, y) \in \mathrm{F}\}$ sowie
$\mathrm{A}\bullet := \{y \mid \exists x \in \mathrm{A} : (x, y) \in \mathrm{F}\}$. ◊

Definition B.3 [Auftragssystem]
Ein *Auftragssystem* $AS = \langle \mathrm{A}, \lessdot \rangle$ besteht aus einer endlichen Menge von Aufträgen (siehe [Jessen und Valk 1987], Definition 1.1.7) und einer irreflexiven, transitiven Relation $(\lessdot) \subseteq \mathrm{A} \times \mathrm{A}$, genannt *Präzedenzrelation* (precedence relation). Für $(a_i, a_j) \in (\lessdot)$ schreiben wir auch $a_i \lessdot a_j$ und nennen den Auftrag a_i *präzedent* zu a_j. a_i heißt *direkt präzedent* zu a_j, wenn $a_i \lessdot a_j$, jedoch $a_i \lessdot a_k \lessdot a_j$ für kein $a_k \in \mathrm{A}$ gilt. Ein *Konnektivitätsgraph* stellt die Relation „direkt präzedent“ als Graph dar. ◊

Definition B.4 [Kausalnetz]
Ein Netz $N = \langle \mathrm{S}, \mathrm{T}, \mathrm{F} \rangle$ heißt *Kausalnetz* (causal net), wenn

- N zyklusfrei ist (siehe [Jessen und Valk 1987], S.26) und
- alle Stellen höchstens eine Eingangs- und Ausgangstransition haben, das heißt:

$$\forall s \in \mathrm{S} : |\bullet s| \leq \text{ und } |s\bullet| \leq 1 \qquad \Diamond$$

Definition B.5 [Markierung]
Es sei $N = \langle \mathrm{S}, \mathrm{T}, \mathrm{F} \rangle$ ein Netz mit der Stellenmenge $\mathrm{S} = \{s_1, \ldots, s_n\}$. Eine *Markierung* (marking) m für N ist eine Verteilung von Marken auf den Stellen, oder formal eine Abbildung

$$\mathrm{m} : \mathrm{S} \to \mathbb{N}$$

von S in die nichtnegativen ganzen Zahlen $\mathbb{N} := \{0, 1, 2, \ldots\}$. $\mathrm{m}(s_i) \in N$ gibt die Anzahl der *Marken* (tokens) in der Stelle s_i an. Mit $\mathrm{M_S}$ oder M bezeichnen wir die Menge aller Markierungen über S. $\Diamond$

Definition B.6 [S/T-Netz]
Ein *Stellen/Transitions-Netz*, kurz *S/T-Netz* (*place/transition net, P/T net*) $N = \langle \mathrm{S}, \mathrm{T}, \mathrm{F}, \mathrm{K}, \mathrm{W}, \mathrm{m_0} \rangle$ besteht aus folgenden Komponenten:

- einem Netz $(\mathrm{S}, \mathrm{T}, \mathrm{F})$ (Definition B.1) mit endlichen Mengen von Stellen S und Transitionen T,
- einer *Kapazitätsfunktion* $\mathrm{K} : \mathrm{S} \to \mathbb{N} \cup \{\omega\}$ ($\mathrm{K}(s) = \omega$ bedeutet keine endliche Kapazität),
- einer *Kantenbewertung* $\mathrm{W} : (\mathrm{S} \times \mathrm{T}) \cup (\mathrm{T} \times \mathrm{S}) \to \mathbb{N}$ mit $\mathrm{W}(x, y) = 0$ genau dann, wenn $(x, y) \notin \mathrm{F}$,
- einer *Anfangsmarkierung* $\mathrm{m_0} : \mathrm{S} \to \mathbb{N}$ mit $\mathrm{m_0}(s) \leq \mathrm{K}(s)$ für $s \in \mathrm{S}$. $\Diamond$

Definition B.7 [B/E-Netz]
Ein *Bedingungs/Ereignis-Netz* (*B/E-Netz, condition/event-net, C/E-net*) $BEN = \langle \mathrm{S}, \mathrm{T}, \mathrm{F}, \mathrm{m_0} \rangle$ sei definiert als S/T-Netz $STN = \langle \mathrm{S}, \mathrm{T}, \mathrm{F}, \mathrm{k}, \mathrm{w}, \mathrm{m_0} \rangle$ mit

(a) $\forall s \in \mathrm{S} : \mathrm{k}(s) = 1$
(jede Stelle hat die Kapazität Eins) und

(b) $\forall (x, y) \in \mathrm{F} : \mathrm{w}(x, y) = 1$
(jede Kante hat das Kantengewicht Eins). $\Diamond$

Definition B.8 [Aktivierung und Schalten in S/T-Netzen]
Es sei $N = \langle \mathrm{S}, \mathrm{T}, \mathrm{F}, \mathrm{K}, \mathrm{W}, \mathrm{m_0} \rangle$ ein S/T-Netz, $t \in \mathrm{T}$ eine Transition und $\mathrm{m_1}, \mathrm{m_2} \in \mathrm{M_S}$ Markierungen.

- t heißt *aktiviert* in $\mathrm{m_1}$, symbolisch $\mathrm{m_1} \xrightarrow[N]{t}$ (activated, firable), falls $\mathrm{m_1}(s) \geq \mathrm{W}(s, t)$ und $\mathrm{m_1}(s) - \mathrm{W}(s, t) + \mathrm{W}(t, s) \leq \mathrm{K}(s)$ für alle $s \in \mathrm{S}$ gilt. (Dabei gelte $n < \omega$ für $n \in \mathbb{N}$.)

- t schaltet m_1 zu m_2, symbolisch $\mathrm{m}_1 \xrightarrow[N]{t} \mathrm{m}_2$ (fires m_1 to m_2), falls t in m_1 aktiviert ist und $\mathrm{m}_2(s) = \mathrm{m}_1(s) - \mathrm{W}(s,t) + \mathrm{W}(t,s)$ für alle $s \in \mathrm{S}$ gilt.

Ist aus dem Kontext deutlich, welches N gemeint ist, so schreibt man auch $\mathrm{m}_1 \xrightarrow{t}$ bzw. $\mathrm{m}_1 \xrightarrow{t} \mathrm{m}_2$. ◊

Definition B.9 [Aktivierung und Schalten von Schritten in S/T-Netzen] Es sei $N = \langle \mathrm{S}, \mathrm{T}, \mathrm{F}, \mathrm{K}, \mathrm{W}, \mathrm{m}_0 \rangle$ ein S/T-Netz, der Schritt $\mathrm{A} \in \mathcal{P}_{\mathrm{M}}(\mathrm{T})$ eine Multimenge von Transitionen und $\mathrm{m}_1, \mathrm{m}_2 \in \mathrm{M}_{\mathrm{S}}$ Markierungen.

- A heißt *aktiviert* in m_1, falls

$$\forall s \in \mathrm{S}, t \in \mathrm{T} : \mathrm{m}_1(s) \geq \sum_{t \in_{\mathrm{M}} \mathrm{A}} \mathrm{A}(t) \cdot \mathrm{W}(s,t).$$

- A schaltet m_1 zu m_2, symbolisch $\mathrm{m}_1 \xrightarrow[N]{\mathrm{A}} \mathrm{m}_2$, falls A in m_1 aktiviert ist und

$$\forall s \in \mathrm{S}, t \in \mathrm{T} : \mathrm{m}_2(s) = \mathrm{m}_1(s) + \sum_{t \in_{\mathrm{M}} \mathrm{A}} \mathrm{A}(t) \cdot (-\mathrm{W}(s,t) + \mathrm{W}(t,s)). \qquad \text{◊}$$

Anhang C

Konstruktion eines Feature-Structure-Prozesses

Dieser Anhang enthält die einzelnen Schritte der Konstruktion eines Prozesses, die in Abschnitt 4.2.8 im einzelnen beschrieben wird.

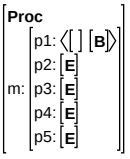

Abbildung C.1: Der initiale Prozeß P_0 des elementaren FSNets aus Abbildung 4.10 als AVM.

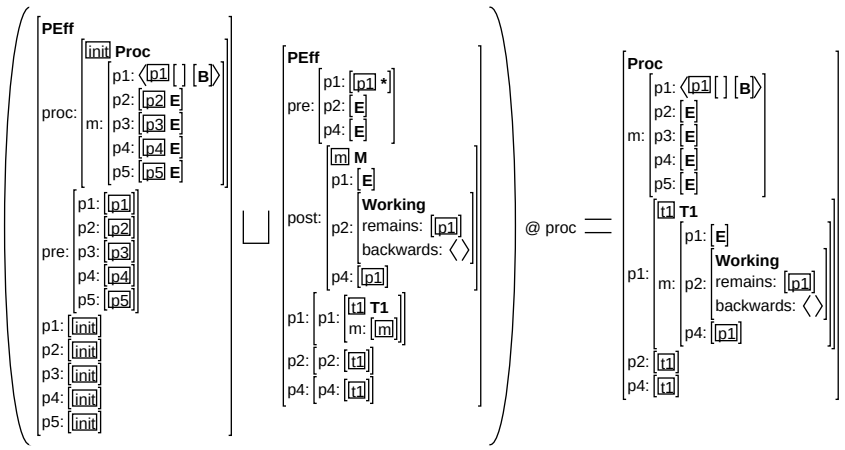

Abbildung C.2: Der maximale S-Schnitt scut(P_0) ergibt mit der Prozeß-Transitionsregel von `t1` den Prozeß P_1.

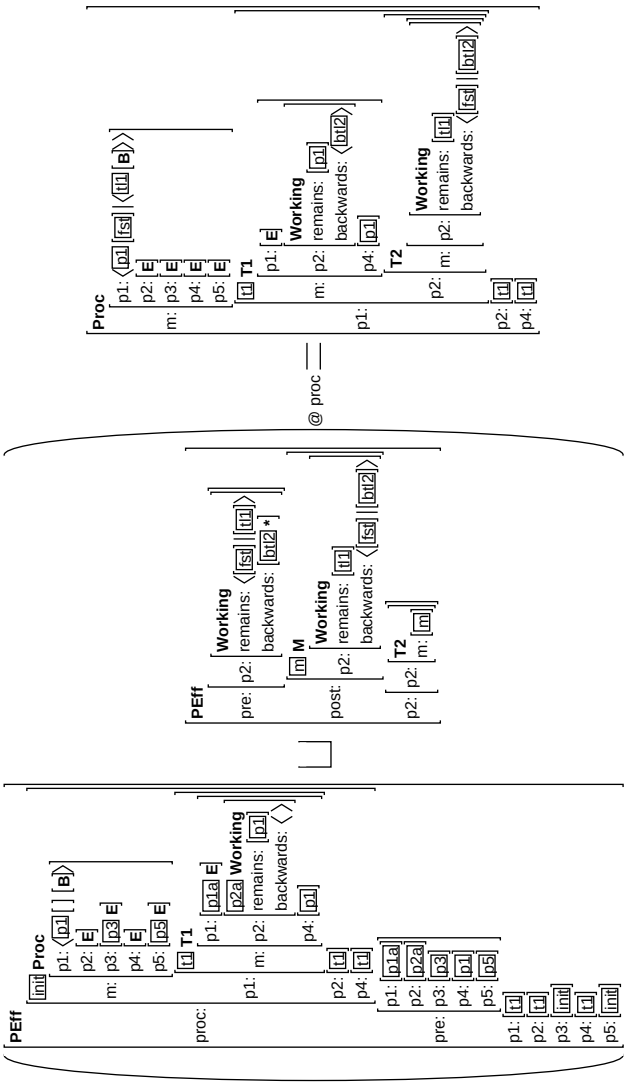

Abbildung C.3: Der maximale S-Schnitt scut(P_1) ergibt mit der Prozeß-Transitionsregel von t2 den Prozeß P_2.

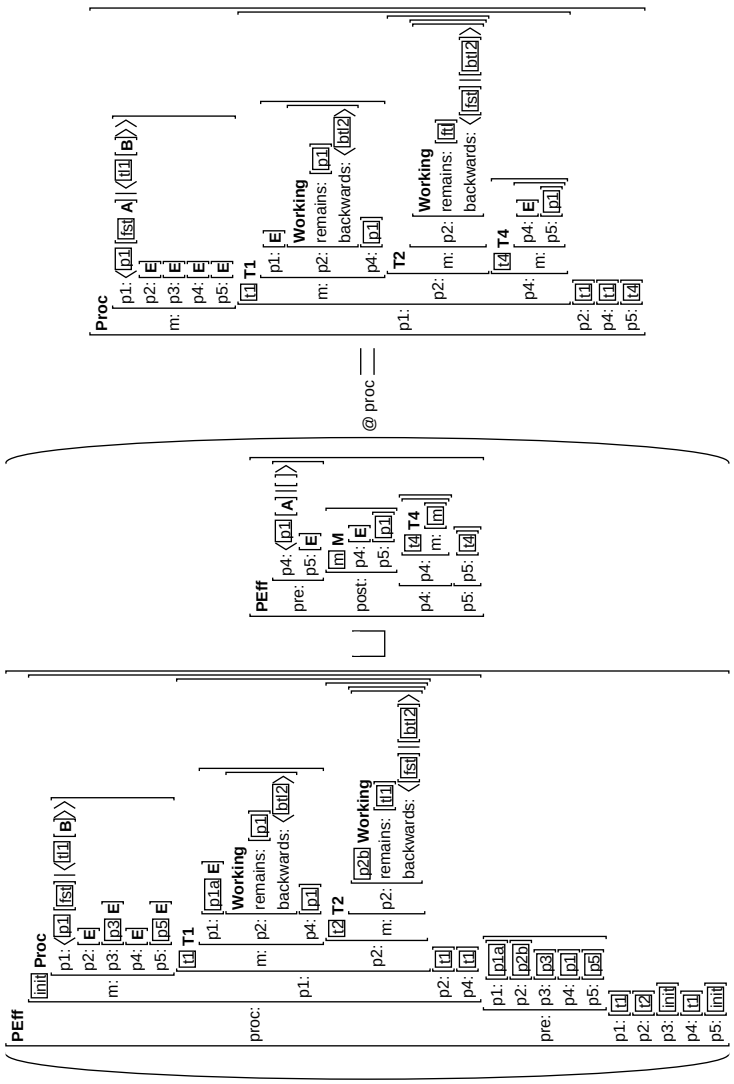

Abbildung C.4: Der maximale S-Schnitt scut(P_2) ergibt mit der Prozeß-Transitionsregel von t4 den Prozeß P_3.

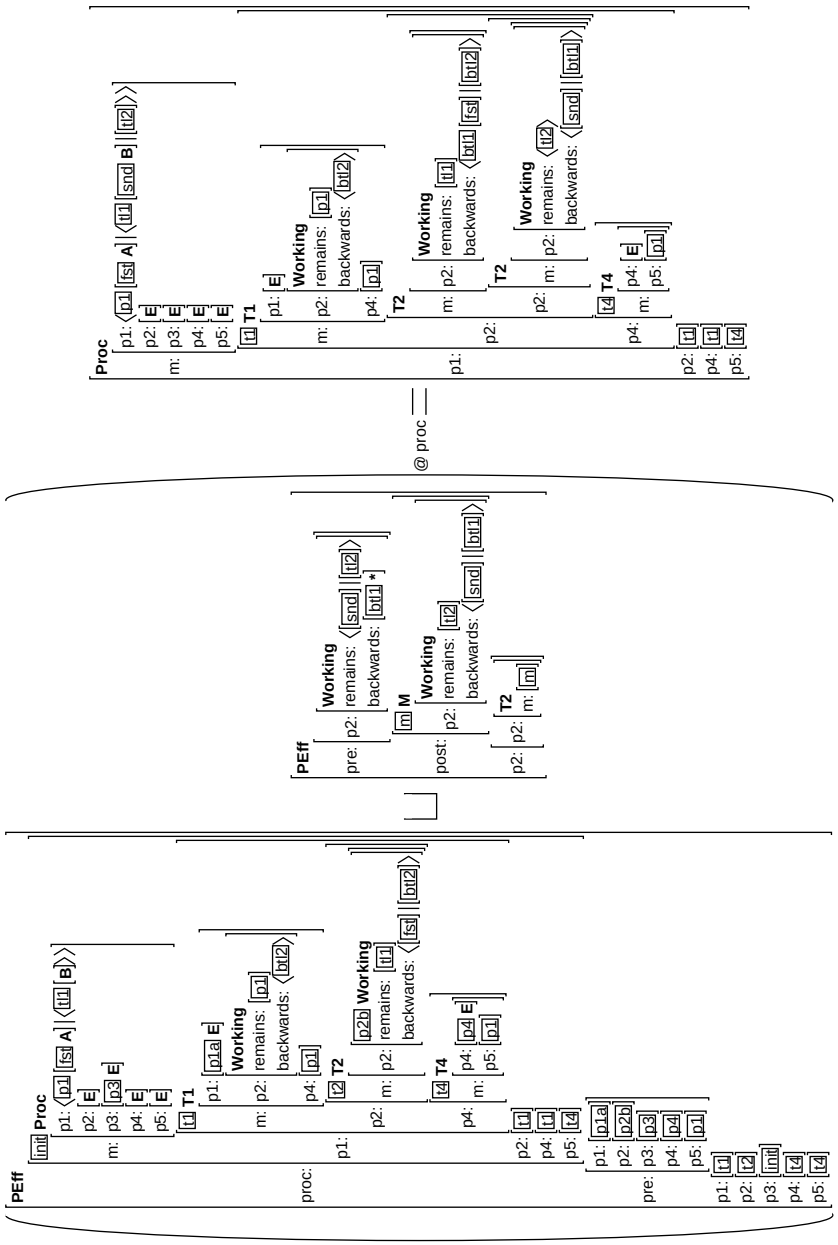

Abbildung C.5: Der maximale S-Schnitt $\mathrm{scut}(P_3)$ ergibt mit der Prozeß-Transitionsregel von t2 den Prozeß P_4.

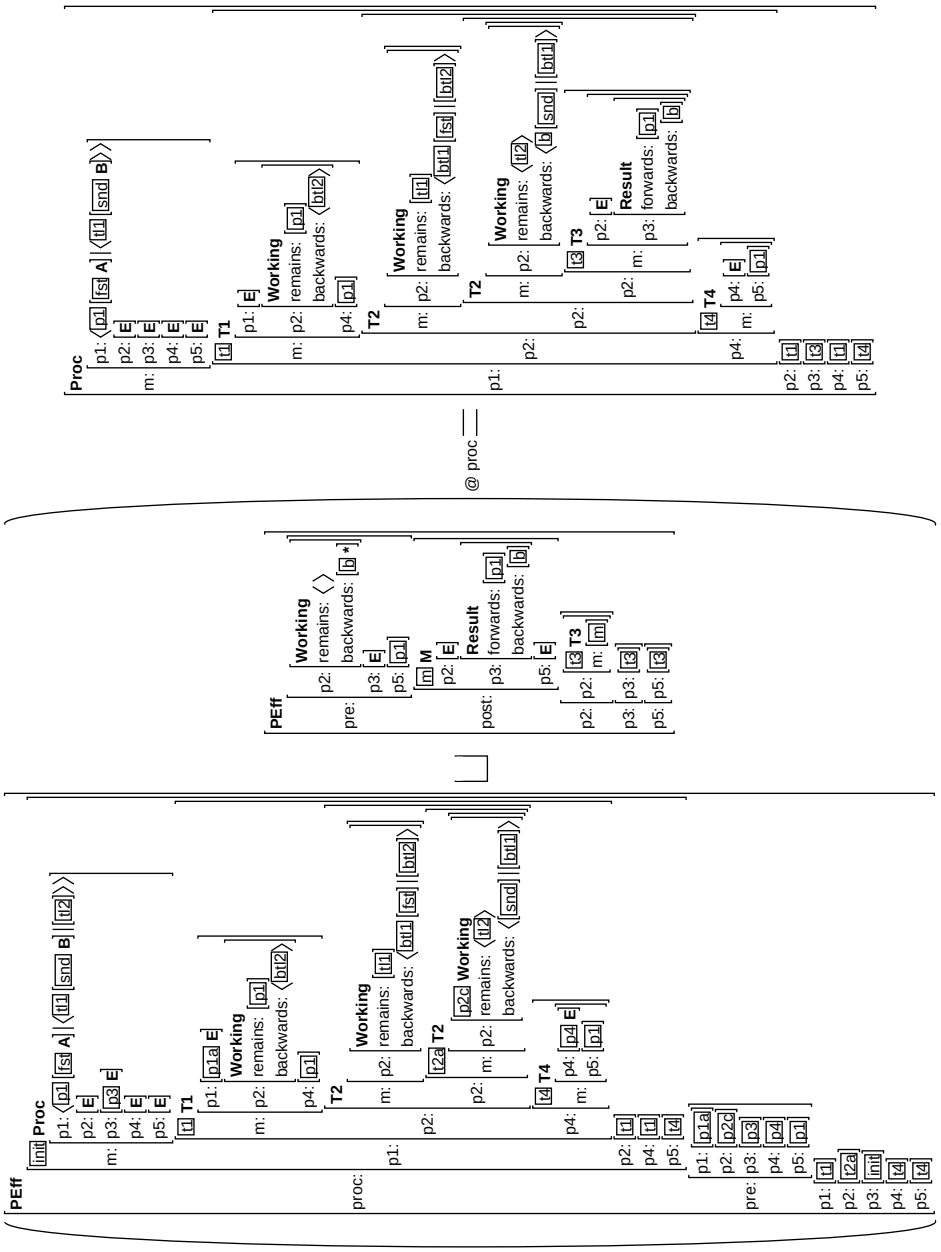

Abbildung C.6: Der maximale S-Schnitt $\mathrm{scut}(P_5)$ ergibt mit der Prozeß-Transitionsregel von t3 den Prozeß P_5.

Literaturverzeichnis

[Aalst et al. 1999] W.M.P. van der Aalst, D. Moldt, R. Valk und F. Wienberg. Enacting Interorganizational Workflow Using Nets in Nets. In J. Becker, M. zur Mühlen und M. Rosemann (Herausgeber), „Workflow Management '99 - Proceedings“, Band 70, Seiten 117–136, Münster (November 1999). Universität Münster, Task Force Workflow.

[Aalst und Anyanwu 1999] W.M.P. van der Aalst und K. Anyanwu. Inheritance of Interorganizational Workflows to Enable Business-to-Business E-commerce. In „Proceedings of the Second International Conference on Telecommunications and Electronic Commerce (ICTEC'99)“, Seiten 141–157. Nashville, Tennessee (Oktober 1999).

[Aalst und Basten 1997] W.M.P. van der Aalst und T. Basten. Life-Cycle Inheritance – A Petri-Net-Based Approach. In Azéma, P. und Balbo, G. (Herausgeber), „18th International Conference on Application and Theory of Petri Nets, Toulouse, France, June 1997“, Band 1248 aus „Lecture Notes in Computer Science“, Seiten 62–81, Berlin (Juni 1997). Springer-Verlag.

[Aalst 1997] W.M.P. van der Aalst. Verification of Workflow Nets. In Pierre Azéma und Gianfranco Balbo (Herausgeber), „Application and Theory of Petri Nets 1997“, Nummer 1248 in Lecture Notes in Computer Science, Seiten 407–426, Berlin; Heidelberg; New York (1997). Springer-Verlag.

[Aalst 1998] W.M.P. van der Aalst. Three Good reasons for Using a Petri-net-based Workflow Management System. *The Kluwer International Series in Engineering and Computer Science: Information and Process Integration in Enterprises: Rethinking Documents, Chapter 10* **428**, 161–182 (1998).

[Aalst 1999a] W. M. P. van der Aalst. Woflan: A Petri-net-based Workflow Analyzer. *Systems Analysis – Modelling – Simulation* **35**(3), 345–357 (1999).

[Aalst 1999b] W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology* **41**(10), 639–650 (1999).

[Aït-Kaci et al. 1992] H. Aït-Kaci, A. Podelski und G. Smolka. A feature-based constraint system for logic programming with entailment. Bericht 92–17, Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern; Saarbrücken (1992).

[Aït-Kaci und Nasr 1986] H. Aït-Kaci und R. Nasr. Login: A logical programming language with built-in inheritance. *Journal of Logic Programming* (3), 187–215 (1986).

[Aoumeur und Saake 1999] N. Aoumeur und G. Saake. Towards an object Petri net model for specifying and validating distributed information systems. *Advanced Information Systems Engineering* **1626**, 381–395 (1999).

[Arnold und Gosling 1996] K. Arnold und J. Gosling. „The Java Programming Language". Addison-Wesley (1996).

[Astesiano et al. 1995] E. Astesiano, G. Reggio und A. Tarlecki (Herausgeber). „Recent trends in data type specification : 10th Workshop on Specification of Abstract Data Types, joint with the 5th COMPASS Workshop, S. Margherita, Italy, May 30-June 3, 1994", Nummer 906 in Lecture Notes in Computer Science, Berlin; New York (1995). Springer-Verlag.

[Bartelt und Lamersdorf 2000] A. Bartelt und W. Lamersdorf. Agent-Oriented Concepts to Foster the Automation of e-Business. In A.M. Tjoa, R.R. Wagner und A. Al-Zobaidi (Herausgeber), „Proceedings of the 11th International Workshop on Database and Expert Systems (DEXA 2000)", Seiten 775–779, Los Alamitos, CA; Washington; Brüssel; Tokyo (2000). IEEE.

[Battiston et al. 1991] E. Battiston, F. de Cindio und G. Mauri. „OBJSA Nets: A Class of High-level Nets Having Objects as Domain", Kapitel 6, Seiten 189–214. In [Jensen und Rozenberg 1991] (1991).

[Baumgarten 1990] B. Baumgarten. „Petri-Netze: Grundlagen und Anwendungen". BI-Wissenschaftsverlag, Mannheim Wien Zürich (1990).

[B.J.Nelson 1981] B.J.Nelson. „Remote Procedure Call". Dissertation, Department of Computer Science, Carnegie Mellon University (1981).

[Blackwell 1996] A.F. Blackwell. Metacognitive Theories of Visual Programming: What do we think we are doing? In „Proceedings IEEE Symposium on Visual Languages", Seiten 240–246 (1996).

[BODTF 1997] BODTF. Combined Business Object Facility. Bericht, Business Object Domain Task Force (1997). BODTF-RFP first Submission.

[Boger et al. 1999a] M. Boger, F. Wienberg und W. Lamersdorf. Dejay: Unifying Concurrency and Distribution to Achieve a Distributed Java. In R. Mitchell, A.C. Wills, J. Bosch und B. Meyer (Herausgeber), „Proceedings of the 29th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'99)“, Seiten 285–294, Nancy, France (1999). IEEE Computer Society Press, Los Alamitos, CA/USA.

[Boger et al. 1999b] M. Boger, F. Wienberg und W. Lamersdorf. Virtual Processors: Migrating Object-Clusters Unify Concurrency and Distribution. *Journal on Integrated Computer-Aided Engineering* (1999).

[Boger et al. 2000] M. Boger, T. Baier, F. Wienberg und W. Lamersdorf. Extreme Modeling. In „Extreme Programming and Flexible Processes in Software Engineering - XP2000“. Addison Wesley (2000).

[Boger 1999] M. Boger. „Java in verteilten Systemem – Nebenläufigkeit, Verteilung und Persistenz“. dpunkt.verlag, Heidelberg (1999).

[Brauer et al. 1987] W. Brauer, W. Reisig und G. Rozenberg (Herausgeber). „Petri Nets – Central Models and their Properties, Advances in Petri Nets“, Band 1 aus „Lecture Notes in Computer Science“. Springer-Verlag, Berlin (1987).

[Brenner et al. 1998] W. Brenner, R. Zarnekow und H. Wittig. „Intelligente Softwareagenten – Grundlagen und Anwendungen“. Springer-Verlag (1998).

[Buchs und Guelfi 1991a] D. Buchs und N. Guelfi. CO-OPN: A Concurrent Object Oriented Petri Net Approach. In „Application and Theory of Petri Nets, 12th International Conference, Gjern, Denmark“, Seiten 432–454. Aarhus University, IBM Deutschland (Juni 1991).

[Buchs und Guelfi 1991b] D. Buchs und N. Guelfi. A Semantic Description of Actor Languages by Structured Algebraic Petri Nets. Bericht 639, Universite de Paris-Sud, L.R.I. (Februar 1991).

[Carpenter und Thomason 1990] B. Carpenter und R. Thomason. „Inheritance theory and path-based reasoning: An Introduction“, Band 5 aus „Studies in Cognitive Systems“, Seiten 309–344. Kluwer Academic Publishing, Dordrecht (1990).

[Carpenter 1990] B. Carpenter. Inheritance, (In)equations, and extensionality. In „Proceedings of the first international Workshop on Inheritance in natural language processing“, Seiten 9–13, Tilburg, Niederlande (1990).

[Carpenter 1992] B. Carpenter. „The Logic of Typed Feature Structures: with applications to unification grammars, logic programs and constraint resolution“.

Nummer 32 in Cambridge tracts in theoretical computer science. Cambridge University Press, New York (1992).

[Castelfranchi und Müller 1993] C. Castelfranchi und J.-P. Müller (Herausgeber). „From Reaction to Cognition – 5th European Workshop on Modelling Autonomous Agents in a Multi-Agent World“, Nummer 957 in Lecture Notes in Artificial Intelligence, Neuchâtel, Switzerland (August 1993). Springer-Verlag.

[Chen und Scheer 1994] P.P.-S. Chen und A.-W. Scheer. Modellierung von Prozeßketten mittels Petri-Netz-Theorie. Bericht 107, Institut für Wirtschaftsinformatik, Saarbrücken (1994).

[Chen 1976] P.P. Chen. The Entity-Relationship Model – Toward a unified view of data. In „ACM Transactions on Database Systems“, Band 1, Seiten 9–36 (1976).

[Christensen und Damgaard Hansen 1992] S. Christensen und N. Damgaard Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. Bericht DAIMI PB–390, Aarhus University (1992).

[Christensen und Damgaard Hansen 1993] S. Christensen und N. Damgaard Hansen. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. In M. Ajmone Marsan (Herausgeber), „Application and Theory of Petri Nets 1993, Proceedings 14th International Conference, Chicago, Illinois, USA“, Band 691 aus „Lecture Notes in Computer Science“, Seiten 186–205. Springer (1993).

`http://www.daimi.au.dk/ sorenchr/publ.html`.

[Ciancarini 1999] P. Ciancarini (Herausgeber). „Formal methods for open object-based distributed systems“. Kluwer Academic Publishing (1999).

[Coad und Yourdon 1991] P. Coad und E. Yourdon. „Object Oriented Analysis“. Prentice Hall, 2. Auflage (1991).

[Conallen 2000] J. Conallen. „Building Web Applications with UML“. Addison-Wesley Object Technology Series. Addison-Wesley Longman, Amsterdam (2000).

[Consortium 2000] Action Semantics Consortium. Action Semantics for the UML – European Homepage (2000). `http://www.kc.com/as_site`.

[Conway 1971] J.H. Conway. „Regular Algebra and Finite Machines.“ Chapman and Hall, London (1971).

[COSA 2000] COSA. Homepage (2000). `http://www.ley.de/cosa`.

[Cosmos-Consortium 1999] Cosmos-Consortium. Cosmos – Component Specification, Deliverable D4. Bericht 26850, European Commission, DG3, ESPRIT (1999).

[Cosmos 1999] Cosmos. Homepage (1999).

`http://vsys-www.informatik.uni-hamburg.de/projects/cosmos`.

[Date 1995] C.J. Date. „An Introduction to Database Systems“. Addison-Wesley, 6. Auflage (1995).

[Design/CPN 2000] Design/CPN. Homepage (2000).

`http://www.daimi.au.dk/designCPN`.

[Douglass 1999] B.P. Douglass. „Doing Hard Time – Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns“. Addison-Wesley Object Technology. Addison-Wesley Longman, Amsterdam (1999). Mit CD-ROM.

[D'Souza und Wills 1998] D. D'Souza und A.C. Wills. „Objects, Components, and Frameworks with UML – The Catalysis Approach“. Addison-Wesley (1998).

[Engels et al. 2000] G. Engels, L. Groenewegen und G. Kappel. Coordinated Collaboration of Objects. In [Papazoglou et al. 2000], Seiten 307–332.

[Evans et al. 2000] A. Evans, S. Kent und B. Selic (Herausgeber). „UML2000 – The Unified Modeling Language; Advancing the Standard. Proceedings of the Third International Conference in York, UK, October 2-6, 2000“, Nummer 1939 in Lecture Notes in Computer Science, Berlin (2000). Springer-Verlag.

[Farwer 1998a] B. Farwer. A Linear Logic View of Object Systems. In H.-D. Burkhard, L. Czaja und P. Starke (Herausgeber), „Informatik-Berichte: Workshop Concurrency, Specification and Programming, 28.–30. September 1998“, Nummer 110 in Informatik-Berichte, Seiten 76–87, Berlin (1998). Humboldt-Universität.

[Farwer 1998b] B. Farwer. Towards Linear Logic Petri Nets – From P/T-Nets to Object Systems. Technischer Report FBI-HH-B-211, Fachbereich Informatik, Universität Hamburg (1998).

[Farwer 1999] B. Farwer. A Linear Logic View of Object Petri Nets. *Fundamenta Informaticae, IOS Press* (37), 225–246 (1999).

[Farwer 2000a] B. Farwer. „Linear Logic Based Calculi for Object Petri Nets“. Logos-Verlag, Berlin (2000).

[Farwer 2000b] B. Farwer. A Multi-region Linear Logic Based Calculus for Dynamic Petri Net Structures. *Fundamenta Informaticae, IOS Press* (43), 61–79 (2000).

[Fowler und Scott 1997] M. Fowler und K. Scott. „UML Distilled – Applying the Standard Object Modeling Language“. Addison-Wesley, Reading, Massachusetts (1997).

[France und Rumpe 1999] R. France und B. Rumpe (Herausgeber). „UML'99 – The Unified Modeling Language; Beyond the Standard. Proceedings of the Second International Conference in Fort Collins, Colorado, USA, October 28-30, 1999“, Nummer 1723 in Lecture Notes in Computer Science, Berlin (1999). Springer-Verlag. `http://www.cs.york.ac.uk/puml/publications.html`.

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson und J. Vlissides. „Design Patterns. Elements of Reusable Object-Oriented Software“. Addison-Wesley (1995).

[Gamma 1998] E. Gamma. „JHotDraw“ (1998).

`http://members.pingnet.ch/gamma/JHD-5.1.zip`.

[Genrich 1987] H.J. Genrich. „Predicate/Transition Nets“, Seiten 208–247. Band 1 aus [Brauer et al. 1987] (1987).

[Georgeff und Rao 1998] M. Georgeff und A. Rao. „Rational Software Agents: From Theory to Practice“, Kapitel 8. Unicom-Serie. Springer-Verlag (1998).

[Giese und Philippi 2000] H. Giese und St. Philippi (Herausgeber). „Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme, 8. Workshop des Arbeitskreises GROOM der GI Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung, 13.-14. November 2000, Universität Münster“ (November 2000). Techreport 24/00-I.

[Ginsberg 1993] M. Ginsberg. „Essentials of Artificial Intelligence“. Morgan Kaufman Publishers (1993).

[Girard 1987] J.-Y. Girard. Linear Logic. *Theoretical Computer Science* **50**(102) (Januar 1987).

[Goldfarb und Prescod 2000] C.F. Goldfarb und P. Prescod. „Das XML Handbuch – Anwendungen, Produkte, Technologien“. Network Computing. Addison-Wesley, 2. erweiterte Auflage (2000). Mit CD-ROM.

[Goldfarb 1991] C.F. Goldfarb. „The SGML handbook“. Oxford University Press (1991).

[Gosling et al. 1996] J. Gosling, B. Joy und G. Steel. „The Java Language Specification“. Addison-Wesley (1996).

[Graaf und Aalst 1998] M.C.A. van de Graaf und W.M.P. van der Aalst. „Systems Engineering: A Petri net based approach to modelling, verification, and implementation“, Kapitel Chapter 27: Workflow Systems., Seiten 531–568. Kronos, Zaragoza (September 1998).

[Grefen et al. 2000] P. Grefen, K. Aberer, Y. Hoffner und H. Ludwig. Cross Flow: Cross-Organizational Workflow Management in Dynamic Virtual Enterprises. *Comput Syst Sci & Eng* **5**, 277–290 (2000).

[Griffel et al. 1998] F. Griffel, M. Boger, H. Weinreich, W. Lamersdorf und M. Merz. Electronic Contracting with Cosmos - How to Establish, Negotiate, and Execute Electronic Contracts on the Internet. In C. Kobryn, C. Atkinson und Z. Milosevic (Herausgeber), „2nd Int. Enterprise Distributed Object Computing Workshop (EDOC'98), San Diego“. IEEE (1998).

[Griffel 1998] F. Griffel. „Componentware. Konzepte und Techniken eines Softwareparadigmas“. dpunkt.verlag, Heidelberg (1998).

[Grossler 2000] O. Grossler. Konzeption und Entwicklung einer Steuerungseinheit zum Ausführen von Verhandlungsnachrichten. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg (2000).

[Häming 1999] A. Häming. Eine COBRA-Infrastruktur zur dynamischen Konfiguration komponentenbasierter Anwendungs-Server. Studienarbeit, Universität Hamburg, Vogt-Kölln Str. 30, 22527 Hamburg (1999).

[Harel und Rumpe 2000] D. Harel und B. Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff. Bericht, The Weizmann Institute of Science, Rehovot, Israel (2000).

[Harel 1987] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* **8-9**, 231–274 (1987).

[Heitsch et al. 2000] S. Heitsch, M. Köhler, M. Martens und D. Moldt. High-level Petri nets for a Model of Organizational Decision Making. In K. Jensen (Herausgeber), „Workshop on the Practical Use of High-Level Petri Nets, HLCPN 2000“, Ny Munkegade, Building 540, DK-8000 Aarhus C, Dänemark (2000). Computer Science Department, Aarhus University.

[Herken 1988] R. Herken (Herausgeber). „The universal turing machine : a half-century survey“, Band 2 aus „Computerkultur“. Kammerer & Unverzagt, Hamburg (1988).

[Heuer 1997] A. Heuer. „Objektorientierte Datenbanken - Konzepte, Modelle, Standards und Systeme“. Addison-Wesley, 2. Auflage (1997).

[Hewitt et al. 1973] C. Hewitt, P. Bishop und R. Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In „Proceedings of the Third International Joint Conference on Artificial Intelligence“, Seiten 235–245 (1973).

[Hitz und Kappel 1999] M. Hitz und G. Kappel. „UML@Work: Von der Analyse zur Realisierung“. dpunkt.verlag, Heidelberg (1999).

[IBM 1996] IBM. FlowMark – Modeling Workflow. Bericht, IBM Deutschland Entwicklungsgesellschaft mbH, Wien (1996). Release 3.

[Jablonski et al. 1997] S. Jablonski, M. Böhm und W. Schulze (Herausgeber). „Workflow-Mangement: Entwicklung von Anwendungen und Systemen; Facetten einer neuen Technologie“. dpunkt.verlag (1997).

[Jablonski und Bußler 1996] S. Jablonski und C. Bussler. „Workflow-Management: Modeling Concepts, Architecture, and Implementation“. International Thomson Computer Press (1996).

[Jacobson et al. 1995] I. Jacobson, M. Ericsson und A. Jacobson. „The Object Advantage, Business Process Reengineering with Object Technology“. Addison-Wesley (1995).

[Jacobson et al. 1999] I. Jacobson, G. Booch und J. Rumbaugh. „The Unified Software Development Process“. Addison-Wesley (1999).

[Jeffrey et al. 1996] J. Jeffrey, J. Lobo und T. Murata. A High-Level Petri Net for Goal-Directed Semantics of Horn Clause Logic. *IEEE Transactions on Knowledge and Data Engineering* **8**(2), 241–259 (April 1996).

[Jensen und Rozenberg 1991] K. Jensen und G. Rozenberg (Herausgeber). „High-level Petri Nets – Theory and Application“. Springer-Verlag, Berlin; Heidelberg; New York (1991).

[Jensen 1990] K. Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark (November 1990).

[Jensen 1992] K. Jensen. „Coloured Petri Nets: Volume 1; Basic Concepts, Analysis Methods and Practical Use“. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin; Heidelberg; New York (1992).

[Jensen 1994] K. Jensen. „Coloured Petri Nets: Volume 2; Analysis Methods“. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin; Heidelberg; New York (1994).

[Jensen 1997] K. Jensen. „Coloured Petri Nets: Volume 3; Practical Use“. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin; Heidelberg; New York (1997).

[Jessen und Valk 1987] E. Jessen und R. Valk. „Rechensysteme: Grundlagen der Modellbildung“. Springer-Verlag, Berlin (1987).

[Kasper und Rounds 1986] R. Kasper und W. Rounds. A Logical Semantics for Feature Structures. In „24th Annual Meeting of the Association for Computational Linguistics (ACL)“, Seiten 257–266, New York (1986). Columbia University Press.

[Keller et al. 1992] G. Keller, M. Nüttgens und A.-W. Scheer. Semantische Prozeßmodellierung auf der Grundlage „Ereignisgesteuerter Prozeßketten (EPK)“. Bericht 89, Institut für Wirtschaftsinformatik, Saarbrücken (1992).

[Kennedy 1999] A. Kennedy. An Overview of the xUML System Development Process. Bericht, Kennedy Carter, 14 The Pines Broad Street, Guildford GU3 3BH, UK (1999). `http://www.kennedycarter.co.uk/download/CTN_75v1_1.pdf`.

[Kent 1997] S. Kent. Constraint diagrams: visualising invariants in OO models. In „Conference proceedings OOPSLA'97“. ACM Press (1997).

[Kinny et al. 1996] D. Kinny, M. Georgeff und A. Rao. A Methodology and Modelling Technique for Systems of BDI Agents. In [Van de Velde und Perram 1996], Seiten 56–71.

[Kummer et al. 1998] O. Kummer, D. Moldt und F. Wienberg. A Framework for Interacting Design/CPN- and Java-Processes. In Jensen, K. (Herausgeber), „Daimi PB-532: Workshop on Practical Use of Coloured Petri Nets and Design/CPN, Aarhus, Denmark, 10-12 June 1998“, Seiten 131–150. Aarhus University (Juni 1998).

[Kummer et al. 1999] O. Kummer, D. Moldt und F. Wienberg. Symmetric Communication between Coloured Petri Net Simulations and Java-Processes. In Donatelli, Susanna und Kleijn, Jetty (Herausgeber), „Application and Theory of Petri Nets 1999, 20th International Conference, ICATPN'99, Williamsburg, VA, USA“, Band 1639 aus „Lecture Notes in Computer Science“, Seiten 86–105. Springer (Juni 1999).

[Kummer et al. 2000a] O. Kummer, A. Laue, M. Liedtke, D. Moldt und H. Rölke. High-Level Petri Netze zur kompakten Modellierung und Implementierung von Workflowanwendungen. In [Giese und Philippi 2000]. Techreport 24/00-I.

[Kummer et al. 2000b] O. Kummer, F. Wienberg und M. Duvigneau. Renew User Guide (2000). http://www.renew.de.

[Kummer et al. 2001] O. Kummer, F. Wienberg und M. Duvigneau. Renew Architecture Guide (2001). http://www.renew.de.

[Kummer und Wienberg 1999] O. Kummer und F. Wienberg. Renew – the Reference Net Workshop. *Petri Net Newsletter* (56), 12–16 (April 1999).

[Kummer und Wienberg 2000] O. Kummer und F. Wienberg. Renew – The Reference Net Workshop Homepage (2000). http://www.renew.de.

[Kummer 1998] O. Kummer. Simulating Synchronous Channels and Net Instances. In Desel, J., Kemper, P., Kindler, E. und Oberweis, A. (Herausgeber), „Forschungsbericht: 5. Workshop Algorithmen und Werkzeuge für Petrinetze“, Nummer 694, Seiten 73–78. Universität Dortmund, Fachbereich Informatik (1998).

[Kummer 1999a] O. Kummer. A Petri Net View on Synchronous Channels. *Petri Net Newsletter* (56), 7–11 (April 1999).

[Kummer 1999b] O. Kummer. Tight Integration of Java and Petri Nets. In „6. Workshop Algorithmen und Werkzeuge für Petrinetze“, Seiten 30–35, Frankfurt am Main (Oktober 1999). J.W. Goethe-Universität, Institut für Wirtschaftinformatik.

[Kummer 2001] O. Kummer. „Referenznetze“. Dissertation, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg (2001). Erscheint Mitte 2001.

[Lakos und Christensen 1993] C.A. Lakos und S. Christensen. A General Systematic Approch to Arc Extensions for Coloured Petri Nets. Technical report R93-7, Department of Computer Science, University of Tasmania, GPO Box 252C, Hobart Tasmania 7001 (August 1993).

[Lakos 1995] C.A. Lakos. From Coloured Petri Nets to Object Petri Nets. In „16th International Conference on the Application and Theory of Petri Nets“, Nummer 935 in Lecture Notes in Computer Science, Seiten 278–297, Torino, Italy (1995). Springer-Verlag.

[Lamersdorf 1994] W. Lamersdorf. „Datenbanken in verteilten Systemen – Konzepte, Lösungen, Standards“. Vieweg, Braunschweig, Wiesbaden (1994).

[Langner et al. 1998] P. Langner, Ch. Schneider und J. Wehler. Petri Nets based Certification of Event-driven Process Chains. In „International Conference on Application and Theory of Petri Nets in Lisbon 1998“, Lecture Notes in Computer Science, Berlin; Heidelberg; New York (1998). Springer Verlag. angenommener Beitrag.

[Larman 1998] C. Larman. „Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design“. Prentice Hall PTR (1998).

[Laue et al. 2000] A. Laue, M. Liedtke, D. Moldt und I. Trickovic. Statecharts as Protocols for Objects. In T. Clark, A. Evans und K. Lano (Herausgeber), „Electronic Workshops in Computing: Third Workshop on Rigorous Object-Oriented Methods ROOM2000“, England (Januar 2000). University of York.

`http://www.ewic.org.uk/ewic/workshop/view.cfm/ROOM2000`.

[Lea 1998] D. Lea. „Overview of the `collections` Package, Version 0.96b“. State University of New York at Oswego (1998).

`http://gee.cs.oswego.edu/dl/classes/collections`.

[Lenz und Oberweis 2001] K. Lenz und A. Oberweis. Modeling Interorganizational Workflows with XML Nets. In „Proceedings of the Hawaii International Conference On System Sciences, January 3-6, 2001, Maui, Hawaii“, 60054 Frankfurt/Main (2001). Institute of Information Systems J.W. Goethe-University.

[Lilius 1996] J. Lilius. Modelling Actors with M-nets. In L. Czaja, P. Starke, H.-D. Burkhard und M. Lenz (Herausgeber), „Workshop on Concurrency, Specification and Programming“, Nummer 69, Humboldt-Universität zu Berlin (1996).

[Louden 1994] K. C. Louden. „Programmiersprachen, Grundlagen, Konzepte, Entwurf“. International Thomson Publishing, 1. Auflage (1994).

[Maibaum und Rumpe 2000] T. Maibaum und B. Rumpe. Automated Software Engineering: Special Issue on Precise Semantics for Software Modeling Techniques (PSMT'- an ICSE'98 Workshop). *Journal of Automated Software Engineering. (PSMT – an ICSE'98 Workshop)* **7** (2000).

`http://www4.informatik.tu-muenchen.de/papers/MR2000.html`.

[Martí-Oliet und Meseguer 1989] N. Martí-Oliet und J. Meseguer. From Petri Nets to Linear Logic. Bericht, SRI International, Computer Science Laboratory, Stanford (1989).

[Mattern 1989] F. Mattern. „Verteilte Basisalgorithmen“. Springer-Verlag, Berlin (1989).

[Matthiessen und Kasper 1987] Ch. Matthiessen und R. Kasper. Representational issues in systemic functional grammar. Bericht, Information Sciences Institute, Marina del Rey, CA (1987).

[Merz et al. 1999a] M. Merz, F. Griffel, M. Boger, H. Weinreich und W. Lamersdorf. Electronic Contracting im Internet. In R. Steinmetz (Herausgeber), „GI/ITG-Konferenz 'Kommunikation in Verteilten Systemen' (KIVS'99)“, Informatik-Aktuell, Seiten 314–325, Berlin, Heidelberg (1999). GI, Springer-Verlag.

[Merz et al. 1999b] M. Merz, B. Libermann und W. Lamersdorf. Crossing organisational boundaries with mobile agents in electronic service markets. *International Journal on Computer-Aided Engineering, Special Issue on 'Mobile Agents'* **6/2**, 91–104 (1999).

[Merz 1999a] M. Merz. „Electronic Commerce – Modelle und Mechanismen des Electronic Commerce“. Springer-Verlag, Berlin; Heidelberg; New York (1999).

[Merz 1999b] M. Merz. „Elektronic Commerce – Marktmodelle, Technologien und Anwendungen“. dpunkt.verlag, Heidelberg (1999).

[Meyer 1997] B. Meyer. „Object-Oriented Software Construction“. Prentice Hall, 2. Auflage (1997).

[MID 2000] MID. Innovator (2000). `http://www.mid.de/products/innovator.php3`.

[Moldt und Rodenhagen 2000] D. Moldt und J. Rodenhagen. Ereignisgesteuerte Prozeßketten und Petrinetze zur Modellierung von Workflows. In [Giese und Philippi 2000]. Techreport 24/00-I.

[Moldt und Wienberg 1997] D. Moldt und F. Wienberg. Multi-Agent-Systems Based on Coloured Petri Nets. In Azéma, P. und Balbo, G. (Herausgeber), „18th International Conference on Application and Theory of Petri Nets, Toulouse, France“, Band 1248 aus „Lecture Notes in Computer Science“, Seiten 407–426, Berlin (Juni 1997). Springer.

[Moldt 1996] D. Moldt. „Höhere Petrinetze als Grundlage für Systemspezifikationen“. Dissertation, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg (August 1996).

[Moshier 1988] D. Moshier. „Extensions to unification grammar for the description of programming languages“. Dissertation, University of Michigan (1988).

[Müller-Wilken et al. 2000] S. Müller-Wilken, F. Wienberg und W. Lamersdorf. On Integrating Mobile Devices Into A Workflow Management Scenario. In A. Al-Zobaidie A. M. Tjoa, R. R. Wagner (Herausgeber), „Proceedings of the 11th International Workshop on Database and Expert Systems Applications (DEXA 2000)“, Seiten 186–192, Hamburg (10 2000). IEEE Computer Society.

[Müller-Wilken und Lamersdorf 2000] S. Müller-Wilken und W. Lamersdorf. JBSA: An Infrastructure for Seamless Mobile Systems Integration. In Claudia Linnhoff-Popien und Heinz-Gerd Hegering (Herausgeber), „Proc. 3rd IFIP/GI International Conference on Trends towards a Universal Service Market (USM 2000)“, Lecture Notes in Computer Science, Seiten 164–175. Ludwig-Maximilians-University Munich, Springer-Verlag (September 2000).

[Müller 1993] J. Müller (Herausgeber). „Verteilte Künstliche Intelligenz – Methoden und Anwendungen“, Band 1. B.I., Mannheim (1993).

[Oasis-Open 2000] Oasis-Open. SGML-Standard (2000).

`http://www.oasis-open.org/cover/`.

[Oberweis et al. 1986] A. Oberweis, F. Schönthaler, G. Lausen und W. Stucky. Net Based Conceptual Modelling and Rapid Prototyping with INCOME. In „Proc. of the 3rd AFCET Conf. on Software Engineering, Versailles“, Seiten 165–176 (1986). NewsletterInfo: 29.

[Oberweis et al. 1987] A. Oberweis, F. Schönthaler, J. Seib und G. Lausen. Database Supported Analysis Tool for Predicate/Transition Nets. *Petri Net Newsletter* (28), 21–23 (Dezember 1987).

[Oberweis et al. 1994] A. Oberweis, G. Scherrer und W. Stucky. Income/Star – methodology and tools for the development of distributed information systems. *Information Systems* **19**(8), 643–660 (1994).

[OMG 2000] OMG. Unified Modeling Language (UML), version 1.3 (2000).

`http://www.omg.org/technology/documents/formal/unified_modeling_language.htm`.

[Papazoglou et al. 2000] M.P. Papazoglou, S. Spaccapietra und Z. Tari (Herausgeber). „Advances in Object-Oriented Data Modeling“, Cooperative information systems, Cambridge, MA (Oktober 2000). MIT Press.

[Papazoglou und Krämer 2000] M.P. Papazoglou und B.J. Krämer. Modelling Object Dynamics. In [Papazoglou et al. 2000], Seiten 195–217.

[Papazoglou 1995] M.P. Papazoglou (Herausgeber). „14th International Conference OOER'95: Object-Oriented and Entity-Relationship Modeling“, Nummer 1021 in Lecture Notes in Computer Science, Gold Coast, Australia (1995). Springer-Verlag.

[Pollard und Sag 1987] C. Pollard und I.A. Sag. „Information-based syntax and semantics“, Band 1: Fundamentals aus „Center for the study of language and information lecture notes“. CSLI, Stanford (1987).

[Pollard und Sag 1994] C.J. Pollard und I.A. Sag. „Head-driven Phrase Structure Grammar“. Studies in contemporary linguistics. Center for the Study of Language and Information, Stanford (1994).

[Precise UML Group 2000] Precise UML Group. Homepage (2000).

`http://www.cs.york.ac.uk/puml`.

[Promatis 2000] Promatis. Income Homepage (2000).

`http://www.get-INCOME.com/products/processdesigner/index.html`.

[Rao und Georgeff 1991] A.S. Rao und M.P. Georgeff. Modelling rational agents within a BDI-architecture. In R. Fikes und E. Sandewall (Herausgeber), „Proceedings of Knowledge Representation and Reasoning“, Seiten 473–484. Morgan Kaufmann Publishers, Inc. (1991).

[Rao 1996] A.S. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In [Van de Velde und Perram 1996], Seiten 42–55.

[Reisig 1986] W. Reisig. „Petrinetze: Eine Einführung“. Springer-Verlag, 2. Auflage (1986).

[Reisig 1990] W. Reisig. „Petri Nets and Algebraic Specifications.“ Nummer 342. Technische Universiät, Institut für Informatik, Sonderforschungsbereich „Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen“, SFB-Bericht/1/90/B, München (März 1990).

[Robbins 1999] J.E. Robbins. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. In „Construction of Software Engineering Tools (CoSET'99)“. University of South Australia (Juni 1999).

[Rölke 1999] H. Rölke. Multi-Agenten-Netze – Modellierung und Implementation eines Multi-Agenten-Systems aus der Basis von Referenznetzen. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg (Dezember 1999).

[Rumbaugh et al. 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy und W. Lorensen. „Object-Oriented Modeling and Design“. Prentice Hall (1991).

[Schill 1992] A. Schill. Remote Procedure Call - Fortgeschrittene Konzepte und Systeme - Ein Überblick. *Informatik Spektrum* (15) (1992).

[Schmid 1993] B. Schmid. Electronic Markets. *Wirtschaftsinformatik* (35), 465–480 (5 1993).

[Schmidt-Schauss 1989] M. Schmidt-Schauss. „Computational aspects of an order sorted logic with term declarations“. Nummer 395 in @LNCS, @LNAI. Springer-Verlag, Berlin, Heidelberg, New York (1989).

[Shanahan 1997] M. Shanahan. „Solving the frame problem: a mathematical investigation of the common sense law of inertia“. Artificial intelligence. MIT Press, Cambridge, MA (1997).

[Shieber 1986] S.M. Shieber. „An Introduction to Unification-Based Approaches to Grammar“, Band 4 aus „CSLI Lecture Notes“. Center for the Study of Language and Information, Stanford (1986).

[Shoham 1990] Y. Shoham. Agent-Oriented Programming. Bericht, Standford University, Stanford, California (1990).

[Shoham 1993] Y. Shoham. Agent-Oriented Programming. *AI* **60**, 51–92 (1993).

[Shoham 1994] Y. Shoham. „Artificial Intelligence Techniques in Prolog“. Morgan Kaufmann, San Francisco, California (1994).

[Shoham 1997] Y. Shoham. „An Overview of Agent-Oriented Programming“, Kapitel 13, Seiten 271–290. AAAI Press. MIT Press, Cambridge, MA (1997).

[Smolka 1988] G. Smolka. A feature logic with subsorts. Bericht 33, IBM Deutschland, Stuttgart (1988).

[Sterling und Shapiro 1994] L. Sterling und E. Shapiro. „The art of prolog – Advanced programming techniques; Logic Programming“. MIT Press, Cambridge, MA, 2. Auflage (1994).

[Strömbäck 1991] L. Strömbäck. Unifying disjunctive feature structures. Bericht, Department of Computer and Information Science, Linköping University, Linköping, Schweden (1991).

[Strömbäck 1992] L. Strömbäck. „Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems“. Dissertation, Department of Computer and Information Science, Linköping University, Linköping, Schweden (1992).

[Sun 2000a] Sun. Homepage (2000). `http://www.sun.com`.

[Sun 2000b] Sun. Javasoft Homepage (2000). `http://www.javasoft.com`.

[Sun 2000c] Sun. XML and Java Homepage (2000). `http://java.sun.com/xml`.

[Tigris.org 2000] Tigris.org. ArgoUML Homepage (2000).

`http://argouml.tigris.org`.

[Tu et al. 1999a] M.T. Tu, F. Griffel und W. Lamersdorf. Integration of Intelligent and Mobile Agents for E-Commerce - A Research Agenda. In St. Kirn und M. Petsch (Herausgeber), „Workshop „Intelligente Softwareagenten und betriebswirtschaftliche Anwendungsszenarien“, Arbeitsbericht“, Band 14. TU Ilmenau, FG Wirtschaftsinformatik 2 (1999).

[Tu et al. 1999b] M.T. Tu, C. Seebode und W. Lamersdorf. A Dynamic Negotiation Framework for Mobile Agents. In „Proceedings of the Third International Symposium on Mobile Agents“, Palm Springs, California (1999). IEEE Computer Society Press.

[Tu et al. 2000a] M.T. Tu, C. Kunze und W. Lamersdorf. A Rule Management Framework for Negotiating Mobile Agents. In „Proc. 4. International Enterprise Distributed Object Computing Conference (EDOC 2000)“, Seiten 135–143. IEEE (2000).

[Tu et al. 2000b] M.T. Tu, C. Seebode und W. Lamersdorf. DynamiCS: An Actor-based Framework for Negotiating Mobile Agents. *Electronic Commerce Research Journal* (2000). Erscheint Anfang 2000.

[Tu et al. 2000c] M.T. Tu, E. Wolff und W. Lamersdorf. Using Genetic Algorithms to Enable Automated Auctions. In K. Bauknecht, S.M. Madria und G. Pernul (Herausgeber), „Proceedings of the 1st International Conference on Electronic Commerce and Web Technologies (EC-Web 2000)“, Lecture Notes in Computer Science, Seiten 389–398. Springer-Verlag (2000).

[Uthmann und Becker 1998] Ch. von Uthmann und J. Becker. Machen Ereignisgesteuerte Prozeßketten (EPK) Petrinetze für die Geschäftsprozeßmodellierung obsolet? *In: EMISA FORUM – Mitteilungen der GI-Fachgruppe 'Entwicklungsmethoden für Informationssysteme und deren Anwendung', Heft 1/1998* Seiten 100–107 (Januar 1998).

[Uthmann 1997] Ch. von Uthmann. Nutzenpotentiale der Petrinetztheorie für die Erweiterung der Anwendbarkeit Ereignisgesteuerter Prozeßketten (EPK). In „Proceedings zum Workshop Formalisierung und Analyse Ereignisgesteuerter Prozeßketten (EPK), Oldenburg, July 1997“, Seiten 1–22 (1997).

[Valk 1987] R. Valk. Modeling of Task-Flow in Systems of Functional Units. Bericht FBI-HH-B-124/87, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg (1987).

[Valk 1991] R. Valk. Modelling Concurrency by Task/Flow EN Systems. In „Proceedings 3rd Workshop on Concurrency and Compositionality“, Nummer 191 in GMD-Studien, St. Augustin, Bonn (1991). Gesellschaft für Mathematik und Datenverarbeitung.

[Valk 1993] R. Valk. Extending S-Invariants for Coloured and Selfmodifying Nets. Fachbereichsbericht FBI-HH-B-165/93, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg (Dezember 1993).

[Valk 1996] R. Valk. On Processes of Object Petri Nets. Fachbereichsbericht FBI-HH-B-185/96, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg (Juni 1996).

[Valk 1998] R. Valk. Petri Nets as Token Objects – An Introduction to Elementary Object Nets. In Desel, J. und Silva, M. (Herausgeber), „19th International Conference on Application and Theory of Petri Nets, ICATPN'98, Lisbon, Portugal, June 1998“, Band 1420 aus „Lecture Notes in Computer Science“, Seiten 1–25, Berlin (Juni 1998). Springer.

[Valk 1999] R. Valk. Reference and Value Semantics for Object Petri Nets. In „Proceedings of Colloquium on Petri Net Technologies for Modelling Communication Based Systems, October 21–22, 1999“, Seiten 169–187. Fraunhofer Gesellschaft, ISST (1999).

[Valk 2000] R. Valk. Relating Different Semantics for Object Petri Nets; Formal Proofs and Examples. Bericht FBI-HH-B-226/00, Universität Hamburg, Fachbereich Informatik, Hamburg (2000).

[Van de Velde und Perram 1996] W. Van de Velde und J.W. Perram (Herausgeber). „Agents Breaking Away – 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World“, Nummer 1038 in Lecture Notes in Artificial Intelligence, Eindhoven, The Netherlands (Januar 1996). Springer-Verlag.

[Verbeek et al. 1999] H. M. W. Verbeek, T. Basten und W. M. P. van der Aalst. Diagnosing Workflow Processes using Woflan. *Computing Science Report 99/02* (1999).

[Verbeek und Aalst 2000] E. Verbeek und W.M.P. van der Aalst. Woflan 2.0: A Petri-Net-Based Workflow Diagnosis Tool. In D. Simpson M. Nielsen (Herausgeber), „Proceedings of the 21st International Conference on Application and Theory of Petri Nets, ICATPN 2000, Aarhus, Denmark, June 2000“, Lecture Notes in Computer Science, Seiten 475–484, Berlin; Heidelberg; New York (2000). Springer-Verlag.

[Vijay-Shanker und Joshi 1988] K. Vijay-Shanker und A. Joshi. Feature structures based tree adjoining grammars. Bericht, Pennsylvania Univ. Philadelphia Moore School of Electrical Engineering, LINC LAB 136 (1988).

[Warmer und Kleppe 1998] J. Warmer und A. Kleppe. „The Object Constraint Language, Precise Modeling with UML“. Addison-Wesley (1998).

[Weitz 1998a] W. Weitz. SGML-Nets: Integrating Document and Workflow Modelling. In „Proceedings of the 31st International Conference on System Sciences, Kohala Coast, Hawaii“, Seiten 185–194, Los Alamitos (Januar 1998). IEEE Computer Society.

[Weitz 1998b] W. Weitz. Workflow Modelling for Internet-Based Commerce – an Approach Based on High-Level Petri Nets. In W. Lamersdorf und M. Merz (Herausgeber), „Trends in Distributed Systems for Electronic Commerce“, Lecture Notes in Computer Science, Seiten 166–178. Springer-Verlag (1998).

[WFMC 1998] WFMC. Terminology & Glossary. Bericht WFMC-TC-1011, Workflow Managment Coalition, Brussels, Belgium (jun 1998). `http://www.aiim.org/wfmc`.

[WFMC 2000] WFMC. Homepage (2000). `http://www.aiim.org/wfmc`.

[Wienberg 1995] F. Wienberg. Implementation eines unifikationsbasierten, nebenläufigen Systems. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg (1995).

[Wienberg 1996] F. Wienberg. Multiagentensysteme auf der Basis Gefärbter Petri-Netze. Diplomarbeit, Universität Hamburg, Vogt-Kölln Str. 30, 22527 Hamburg (1996).

[Wooldridge und Jennings 1995] M. Wooldridge und N.R. Jennings. Agent Theories, Architectures, and Languages: A Survey. In M.J. Wooldridge und N.R. Jennings (Herausgeber), „Intelligent Agents – ECAI-94 Workshop on Agent Theories, Architectures, and Languages“, Nummer 890 in Lecture Notes in Artificial Intelligence, Seiten 1–21, Amsterdam, The Netherlands (August 1995). Springer-Verlag.

[Zeller und Snelting 1996] A. Zeller und G. Snelting. Unified versioning through feature logic. Bericht, Inst. für Programmiersprachen und Informationssysteme, Abt. Softwaretechnologie, Technische Universität Braunschweig (1996).

[Zeller und Snelting 1997] A. Zeller und G. Snelting. Unified Versioning Through Feature Logic. *ACM Transactions on Software Engineering and Methodology* **6**(4), 398–441 (Oktober 1997).

[Zeller 1994] A. Zeller. Configuration management with feature logics. Bericht, Inst. für Programmiersprachen u. Informationssysteme, Arbeitsgruppe Softwaretechnologie, Technische Universität Braunschweig (1994).