Lehrstuhl IV Software & Systems Engineering
Institut für Informatik
der Technischen Universität München

Ein Produktmodell für die Entwicklung verteilter Informationssysteme

Wolfgang Schwerin

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. P. P. Spies

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. M. Broy
2. Univ.-Prof. Dr. M. Wirsing, Ludwig-Maximilians-Universität München

Die Dissertation wurde am 20.08.2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 24.06.2004 angenommen.

.

Zusammenfassung

Die steigende Komplexität von Softwarelösungen macht ein schrittweises Vorgehen bei der Entwicklung notwendig. Daher ist ein klar definiertes Vorgehensmodell unverzichtbar, das durchzuführende Aktivitäten und zu erbringende Ergebnisse, die häufig auch als (Entwicklungs-)Produkte bezeichnet werden, angibt.

Die Anwendung existierender Vorgehensmodelle wird teilweise dadurch erschwert, dass Inhalte und Zusammenhänge von Produkten nicht immer klar definiert sind. Insbesondere Bezüge zwischen Produkten unterschiedlicher Entwicklungsaufgaben sind vielfach nicht transparent. Zudem wird kaum auf die besonderen Anforderungen eingegangen, die sich für die Entwicklung verteilter Softwaresysteme ergeben. Im Unterschied dazu stellt die Informatik eine große Bandbreite präziser mathematischer Modellierungstechniken zur Verfügung. Diese sind jedoch meist nicht umfassend, sondern auf bestimmte Entwicklungsaufgaben, wie die Spezifikation von Datentypen oder von Verhaltenseigenschaften, beschränkt.

In dieser Arbeit wird eine mathematisch fundierte Grundlage für durchgängige Vorgehensmodelle zur Entwicklung verteilter Informationssysteme geschaffen. Hierzu werden in einem Produktmodell die Arten von Produkten integriert, die im Rahmen der Unternehmens- und der Anforderungsspezifikation sowie des Softwaresystem-Entwurfs zu erbringen sind. Produktinhalte und -zusammenhänge werden präzise anhand eines mathematischen Systemmodells formuliert.

Das Systemmodell ist ein integriertes Modell aller Inhalte der drei behandelten Entwicklungsaufgaben. Es umfasst ein für die Unternehmensspezifikation geeignetes, an zustandskapselnden Objekten und Abläufen orientiertes Modell fachlicher Aufgabenstellungen. Die Inhalte des Softwaresystem-Entwurfs werden im Systemmodell durch ein auf Komponenten und Interaktion basierendes Modell verteilter Systeme abgedeckt. Darüber hinaus enthält das Systemmodell Elemente, anhand derer beschrieben werden kann, wie Abläufe und Objekte durch Komponenten und deren Interaktion realisiert werden. Auf diese Elemente wird in der Anforderungsspezifikation Bezug genommen.

Sowohl im Produkt- als auch im Systemmodell werden bewährte mathematische Modellierungstechniken aufgegriffen, angepasst und zu einem Ansatz vervollständigt, der von der Spezifikation fachlicher Datentypen und von Geschäftsprozessen bis hin zum Feinentwurf von Softwarekomponenten reicht. Damit leistet die Arbeit einen wesentlichen Beitrag, die Lücke zwischen mathematischen Modellierungstechniken und umfassenden Vorgehensmodellen der Informatik zu schließen.

Danksagung

Für die Ermutigung zur Beschäftigung mit dem Themengebiet der vorliegenden Arbeit, die Mithilfe bei der Themenfindung und die Unterstützung bei der Durchführung bedanke ich mich bei Prof. Dr. Manfred Broy. Bei Prof. Dr. Martin Wirsing bedanke ich mich für die Übernahme des zweiten Gutachtens und die Durchsicht einer Vorversion meiner Arbeit.

Für zahlreiche Diskussionen bedanke ich mich bei Bernhard Deifel, Christian Salzmann, Bernhard Schätz, Katharina Spies und Bernhard Weiß. Besonderer Dank gilt Bernhard Deifel, Katharina Spies, Bernhard Weiß und meinem Vater für die sorgfältige Durchsicht und die hilfreichen Kommentare zu einer Vorversion dieser Arbeit.

Nicht minder wichtig für das Entstehen dieser Arbeit war die geduldige Nachsicht und Aufmunterung meiner Familie, Freunde und Kollegen. Für ihre Unterstützung bedanke ich mich sehr herzlich.

Inhaltsverzeichnis

Inhaltsverzeichnis i

1 Einleitung **1**

1.1 Motivation ... 1
1.2 Zielsetzung ... 4
1.3 Vorgehensweise ... 10
1.3.1 Integration ... 14
1.4 Ergebnisse ... 17
1.5 Verwandte Arbeiten ... 20
1.6 Aufbau der Arbeit ... 22
1.7 Durchgängiges Beispiel ... 23

2 Systemmodell **25**

2.1 Einleitung ... 25
2.2 Systemmodell, Systeme und Entwicklungsprodukte ... 29
2.3 Ganzheitliche Perspektive ... 39
2.3.1 Entitäten und Zustandsraum der ganzheitlichen Perspektive ... 41
2.3.2 Prozesse ... 54
2.3.3 Reaktionsabbildung ... 58
2.4 Eigenschaften der ganzheitlichen Perspektive ... 60
2.4.1 Identifikatorsorten zur Abbildung von Entitätsrelationen ... 61
2.4.2 Entitäten: Kapselung und Verbergen von Information ... 62
2.4.3 Synchronisation und Lokalität von Ereignismarkierungen ... 63
2.5 Komponentenbasierte Perspektive ... 65
2.5.1 Überblick ... 65
2.5.2 Realisierungsnetzwerk ... 70
2.5.3 Topologiekomponenten und -netzwerke ... 71
2.5.4 Spezifikationsnetzwerke ... 77
2.5.5 Komponentenautomaten ... 88
2.5.6 Globaler Netzwerkzustand ... 91
2.5.7 Reaktionsabbildung ... 92

2.6 Perspektiven-Zusammenhänge 94
2.6.1 Zustandsrealisierung 95
2.6.2 Verhaltensrealisierung 100
2.6.3 Verwandte Arbeiten 106

3 Datenspezifikation 109
3.1 Reduktspezifikationen 111
3.2 Umfassende Spezifikationen 121
3.3 Entitätsspezifikation 122
3.4 Techniken der Datenspezifikation 123
3.4.1 Algebraische Spezifikation 124
3.4.2 Algebraische Spezifikation objektorientierter Klassen 125
3.4.3 Konstruktive Techniken 127

4 Prozessspezifikation 129
4.1 Geschäftsprozess-Spezifikation 131
4.1.1 Kausalität und Fairness 143
4.2 Sicherheitseigenschaften 146
4.3 Prozessspezifikation und temporale Logik 149

5 Komponentenbasierte Spezifikation 151
5.1 Statische Struktur 152
5.1.1 Nachrichten und Kanäle 154
5.1.2 Spezifikationsnetzwerke 155
5.1.3 Syntaktische Schnittstellen 157
5.1.4 Topologische Klassifikation 159
5.2 Netzwerk-Verhalten 162
5.2.1 Reaktionsspezifikation 164
5.2.2 Kausalität und Fairness 169
5.2.3 Invariantenspezifikation 171
5.3 Automatensicht einer Komponente 171
5.4 Interaktionsverhalten einer Komponente 183
5.5 Realisierungsbeziehung 184
5.5.1 Zustandsrealisierung 184
5.5.2 Ereignisrealisierung 185

6 Komponierte Entwicklungsprodukte 191
6.1 Spezifikation der fachlichen Aufgabenstellung 193
6.1.1 Fachliche Daten- und Zustandssicht 194
6.1.2 Prozesssicht 196
6.2 Spezifikation der Systemgrenze 199

6.2.1 Partitionierung der Entitäten 200
6.2.2 Klassifikation der Ereignismarkierungen 202
6.2.3 Nutzungsspezifikation des Softwaresystems....207
6.3 Anforderungsspezifikation 212
6.3.1 Verwandte Produktarten....216
6.4 Architekturen....218

7 Ausblick 239

8 Grundlegende Definitionen 243
8.1 Allgemeine Grundlagen 243
8.1.1 Ausgezeichnete Mengen....243
8.1.2 Relationen....244
8.1.3 Schreibweisen für Funktionen....244
8.1.4 Prädikatenlogik....245
8.1.5 Sequenzen und Ströme 245
8.2 Signaturen und Algebren....246
8.3 Prozesse....251
8.4 Komponenten, Netzwerke und Automaten 256
8.4.1 Komponenten 257
8.4.2 Interaktions-Zustands-Tupel und -Ströme....262
8.4.3 Automaten 266

Literaturverzeichnis 271

1 Einleitung

1.1 Motivation

Große und weiter zunehmende Funktionsumfänge sowie inhärente Verteilung zeichnen aktuelle und zukünftige Softwarelösungen aus. Beide Charakteristika bedeuten eine hohe Komplexität der Softwaresysteme und ihrer Entwicklung. Zudem stellt die Informationstechnologie mit wachsendem Anteil eine der Schlüsseltechnologien von Produkten und Dienstleistungen dar, und ist somit entscheidend für die Produktivität und den Erfolg von Unternehmen. Es ist damit unverzichtbar, qualitativ hochwertige Softwarelösungen effizient zu entwickeln. Voraussetzung hierfür sind Vorgehensmodelle, die klare Leitlinien für einen systematischen Entwicklungsprozess geben.

Ein bewährtes Prinzip für die Umsetzung komplexer Aufgabenstellungen ist ein schrittweises Vorgehen. Das Endergebnis ergibt sich hierbei durch sukzessive Erstellung von aufeinander aufbauenden Teilergebnissen. Teilergebnisse sind dadurch gekennzeichnet, dass sie nur einige Inhalte des Endergebnisses erfassen und nicht alle Einzeleinheiten dieser Inhalte angeben. Zusätzlich kann es sinnvoll sein, bestimmte Inhalte auf unterschiedliche Weise zu beschreiben.

Beispielsweise kann die schrittweise Entwicklung einer Softwarelösung mit der Spezifikation der zu verwaltenden fachlichen Daten und der zu unterstützenden Geschäftsprozesse beginnen. In einem nächsten Schritt kann die statische Komponentenstruktur der Softwarelösung formuliert werden. Aufbauend darauf wird dann zum Beispiel eine erste Verhaltensspezifikation erstellt, die in Folgeschritten bis hin zur Implementierung verfeinert wird. Verhalten kann dabei sowohl in Form von Interaktionsszenarien als auch in Form von Automaten beschrieben werden.

Somit entstehen in einer Softwareentwicklung durch ein schrittweises Vorgehen eine Vielzahl von Ergebnissen, die sich anhand

- der adressierten Inhalte (zum Beispiel fachliche Daten oder Verhalten der Softwarelösung),
- des Detaillierungsgrades sowie

- der Form der Beschreibung (zum Beispiel Interaktionsszenarien oder Automaten zur Verhaltensspezifikation)

unterscheiden können. Aufgrund dieser Vielfalt von Entwicklungsergebnissen stellen sich Fragen nach ihrer Konsistenz. Um diese beantworten zu können, müssen sowohl die Inhalte einzelner Ergebnisse als auch deren Zusammenhänge klar definiert sein. Hierzu sind Ergebnisse zu integrieren. Für eine effiziente Entwicklung ist zudem die Abstimmung der zu erbringenden Ergebnisse notwendig, so dass bruchfreie Übergänge und damit ein systematisches Vorgehen ermöglicht werden.

An diesen Punkten setzen Vorgehensmodelle zur Softwareentwicklung, die häufig auch als Softwareentwicklungsprozesse und -methoden bezeichnet werden, an, indem sie das Vorgehen in aufeinander abgestimmte Entwicklungsschritte und -ergebnisse gliedern.

Beispielsweise umfasst eine Entwicklung nach dem V-Modell [BDI97] unter anderem die aufeinander aufbauenden Schritte der System-Anforderungsanalyse, des System-Entwurfs sowie des Software-Grobentwurfs. Ein Ergebnis der System-Anforderungsanalyse sind die sogenannten Anwenderforderungen, welche die vom Anwender gestellten fachlichen Anforderungen an das zu entwickelnde System enthalten. Sie dienen als Eingabe für den System-Entwurf, dessen zentrales Ergebnis die sogenannte System-Architektur darstellt. Die Anwenderforderungen und die Systemarchitektur einer Entwicklung werden nur dann als konsistent angesehen, wenn alle fachlichen Anforderungen in der Architektur realisiert sind.

Im Folgenden verwenden wir den Begriff des Entwicklungsproduktes[1] synonym zu Entwicklungsergebnis sowie den Begriff der Entwicklungsaktivität synonym zu Entwicklungsschritt. Das allgemeine Ziel von Vorgehensmodellen ist es, bewährtes Entwicklungswissen in Form von Leitlinien zur Verfügung zu stellen, um dadurch die Effektivität und Effizienz der tatsächlich durchgeführten Entwicklungsprozesse sowie die Qualität der Ergebnisse zu steigern. Folgendes Zitat aus [DAW99] unterstreicht die Bedeutung des Entwicklungsprozesses:

> *„This emphasis on* process *provides the main justification of many standardisation initiatives, as well as of the efforts to measure process maturity [...]. These convergent efforts reflect an evolution of the concept of software quality from the traditional* verification *and* validation *(V&V) approach towards process-focused environments. Underlying this shift is the conviction that quality improvement and cost reduction are best served by pre-certifying processes and then monitoring that these processes are adhered to. Increasing the probability that software is built right in the first place is better than redoing work when V&V reveals faults."*
>
> [DAW99]

Um eine Vorstellung davon zu geben, was wir unter einem Vorgehensmodell verstehen, gehen wir kurz auf die große Bandbreite existierender Ansätze ein (vergleiche etwa [DAW99]), die wir nach folgenden Punkten unterscheiden können:

---

[1] im Englischen häufig als *work product* bezeichnet.

- **Fokussierung auf unterschiedliche Vorgehensmodellelemente.**

  Während Vorgehensmodelle, wie beispielsweise ISO-12207 [ISO95], den Schwerpunkt auf die Entwicklungsaktivitäten legen, stehen in anderen Ansätzen, beispielsweise dem Unified Process [JBR99] und OOTC [IBM97], Entwicklungsprodukte mit den einzusetzenden Modellierungskonzepten und Notationen im Vordergrund. Strategien für die Komposition von Aktivitäten zu einem Entwicklungsprozess sind das zentrale Element von Ansätzen wie dem Wasserfall- [Roy70] und dem Spiralmodell [Boe87].

- **Unterschiedliche Detaillierungsgrade.**

  Allgemein gehaltene Vorgehensmodelle beschränken sich auf die Angabe von Schlüsseleigenschaften von Entwicklungsaktivitäten und -produkten, so dass für die Anwendung dieser Modelle eine organisations-, systemklassen- und projektspezifische Konkretisierung notwendig ist. Ein Beispiel hierfür ist der Entwicklungsstandard ISO-12207 [ISO95]. Auf der anderen Seite stehen Vorgehensmodelle, die Entwicklungsprodukte und zugehörige Notationen detailliert charakterisieren sowie darauf abgestimmte Entwicklungsaktivitäten vorgeben. Beispiele hierfür sind Cleanroom [PTL+99] und Catalysis [DW99], welche ohne weitere Konkretisierung angewandt werden können. Zwischen diesen beiden Polen liegt etwa der deutsche Standard, das V-Modell [BDI97], der zu einem allgemein gehaltenen Rahmen, Vorschläge für Konkretisierungsalternativen anbietet.

- **Unterschiedliche Zielsetzungen.**

  Zielsetzung aller oben genannten Vorgehensmodelle ist es, Leitlinien für die Durchführung von Entwicklungs- und Managementaufgaben anzugeben. Im Unterschied beziehungsweise ergänzend dazu, bieten Ansätze wie das Capability Maturity Model for Software [PCC+93], Bootstrap [Kuv94] und SPICE [Rou95], Unterstützung bei der Bewertung und Verbesserung von Entwicklungsprozessen.

Viele der existierenden Vorgehensmodelle sind informell und zumindest teilweise unpräzise beschrieben. Daher ist unter anderem die angesprochene Integration von Entwicklungsergebnissen nicht immer in ausreichendem Maße gegeben. Zudem sind sie nicht auf die speziellen Anforderungen ausgerichtet, die sich bei der Entwicklung *verteilter* Systeme ergeben.

Im Unterschied dazu stehen bewährte, mathematische Modellierungstechniken und -methoden der Informatik für die diversen Aufgaben einer Softwareentwicklung zur Verfügung. Der Vorteil dieser mathematischen Ansätze ist deren präzise Formulierung, wodurch etwa Inhalte und Zusammenhänge von Entwicklungsergebnissen klar definiert sind. Zudem bieten sie Techniken zur Überprüfung von Fragen der Konsistenz und der Korrektheit an. Inzwischen ist auch der Zugang zu diesen formalen Methoden erleichtert, da praxisnahe, teilweise graphische Spezifikationsnotationen existieren. Jedoch sind diese Ansätze auch durch die Beschränkung auf bestimmte Entwicklungsaufgaben charakterisiert, so dass sie jeweils nur Teile einer Softwareentwicklung abdecken. Beispielsweise sind Methoden der algebraischen Spezifikation, wie etwa LARCH [GH93], primär auf die Datenspezifikation ausgerichtet. Andere Ansätze, wie CCS [Mil89], CSP [Hoa85], FOCUS [BS00], TLA [Lam94] und Unity [CM88], dienen dagegen der Verhaltensspezifikation.

Aus obigen Betrachtungen folgern wir:

- Um die Handhabbarkeit und Qualität von Vorgehensmodellen zu verbessern, ist die *Präzisierung* und *Konkretisierung* ihrer Inhalte, insbesondere im Hinblick auf die Entwicklung verteilter Softwaresysteme, notwendig.
- Um den Einsatz existierender, mathematischer Ansätze zu erleichtern und zu systematisieren, sind diese zu einem *umfassenden* Ansatz zu integrieren.

Beides erreichen wir, wenn wir umfassende Vorgehensmodelle durch Verwendung und Integration mathematischer Modellierungstechniken konkretisieren und präzisieren.

1.2 Zielsetzung

Wie oben dargestellt, können wir zu verbesserten Vorgehensmodellen kommen, wenn wir darin die Aufgaben einer Softwareentwicklung umfassend, konkret und präzise erfassen. Daraus ergibt sich die Zielsetzung dieser Arbeit:

> *Durch die Verwendung und Integration bewährter, mathematischer Modellierungstechniken der Informatik, geben wir ein Modell von Entwicklungsprodukten an, das eine geeignete Basis für die schrittweise und bruchfreie Entwicklung verteilter Informationssysteme darstellt. Wir decken die Entwicklungsaufgaben*
>
> - *Spezifikation einer fachlichen Aufgabenstellung / Unternehmensspezifikation*
> - *Anforderungsspezifikation sowie*
> - *Softwaresystem-Entwurf*
>
> *ab. Die Form der eingeführten Arten von Entwicklungsprodukten sowie die Auswahl der eingesetzten mathematischen Modellierungstechniken orientieren wir an in der Praxis relevanten Vorgehensmodellen und Entwicklungsmethoden.*

Vor einer weiteren Beschreibung der Zielsetzung geben wir unser Verständnis einer Reihe zentraler Begriffe dieser Arbeit an:

Entwicklungsprodukt.
Entwicklungsprodukte, oder kurz Produkte, stellen die Ergebnisse einer Entwicklung dar. Durch Entwicklungsprodukte beschreiben wir Entwicklungsinhalte, wobei wir unter einem **Entwicklungsinhalt** jegliche, für eine Entwicklung bedeutsame Information verstehen. Zum Beispiel sind Topologie und Verhalten eines Softwaresystems die Entwicklungsinhalte eines Produktes, dass wir Software-Architektur nennen können.

Produktart.
Mit einer Produktart geben wir ein Schema zur Beschreibung von Entwicklungsinhalten an. Daher kann durch geeignete Produktarten eine sinnvolle Strukturierung der Ergebnisse einer Entwicklung unterstützt werden. Beispielsweise gibt die Pro-

duktart der Automatenspezifikation einer Komponente ein Schema zur Beschreibung von Komponentenverhalten vor, das aufgrund seines konstruktiven Charakters schematisch in eine programmiersprachliche Implementierung umzusetzen und somit für den Feinentwurf von Komponenten geeignet ist.

Produktmodell.
Unter einem Produktmodell verstehen wir eine integrierte Menge von Produktarten, durch welche die Form der im Rahmen einer Entwicklung zu erbringenden Ergebnisse bestimmt ist. Auf den Aspekt der Integration von Produkten gehen wir in Abschnitt 1.3.1 detailliert ein.

Aktivität.
Im Rahmen einer Entwicklung sind Aktivitäten auszuführen. Die Ausführung einer Aktivität dient im allgemeinen dazu, ausgehend von vorliegenden Produkten diese zu bearbeiten und/oder weitere Produkte zu entwickeln. Ein Beispiel ist die oben erwähnte Aktivität des System-Entwurfs, die Element des V-Modells [BDI97] ist. Sie ist, unter Bezugnahme auf die Produktart der sogenannten Anwenderforderungen und der Systemarchitektur, wie folgt charakterisiert:

> *„Auf der Basis der fachlichen Anforderungen und der Randbedingungen (Anwenderforderungen) wird ein Lösungsvorschlag für eine mögliche technische Struktur des Systems (Systemarchitektur) erarbeitet. Dabei ist besonders auf den geeigneten Einsatz von Fertigprodukten zu achten. [...].“* [BDI97]

Eine Aktivität ist somit durch die Produkte charakterisiert, die vorliegen müssen um die Aktivität ausführen zu können, sowie durch die Produkte, welche durch Ausführung der Aktivität entwickelt werden. Wir sprechen vom Eingabe- und Ergebnis-Kontext einer Aktivität.

Entwicklungsprozess.
Ein Entwicklungsprozess gibt die in einer Entwicklung auszuführenden Aktivitäten sowie Regeln für die Ausführungsreihenfolge an.

Entwicklungsaufgabe.
Wir verwenden den Begriff der Entwicklungsaufgabe, um eine Softwareentwicklung grob zu gliedern. Eine Aufgabe ist durch ihr Ziel charakterisiert und wird durch Ausführung von Aktivitäten erfüllt.

Die Zusammenhänge zwischen obigen Begriffen illustriert Abbildung 1.1 in Form eines UML-Klassendiagramms [OMG01].

Wie eingangs des Kapitels erwähnt, sind die verschiedenen Arten zu erbringender Entwicklungsprodukte und durchzuführender Aktivitäten die Inhalte von Vorgehensmodellen. In dieser Arbeit konzentrieren wir uns auf Entwicklungsprodukte, da diese grundlegend für die Formulierung von Aktivitäten und damit von aus Aktivitäten komponierten Entwicklungsprozessen sind. *Ein* Modell von Entwicklungsprodukten stellt im allgemeinen die Grundlage für eine *Vielzahl* unterschiedlicher Prozessarten dar. Dies wird daran deutlich, dass sogar in Forward- und Reverse-Engineering-Prozessen, zumindest im Kern, dieselben Entwicklungsprodukte zu erstellen sind.

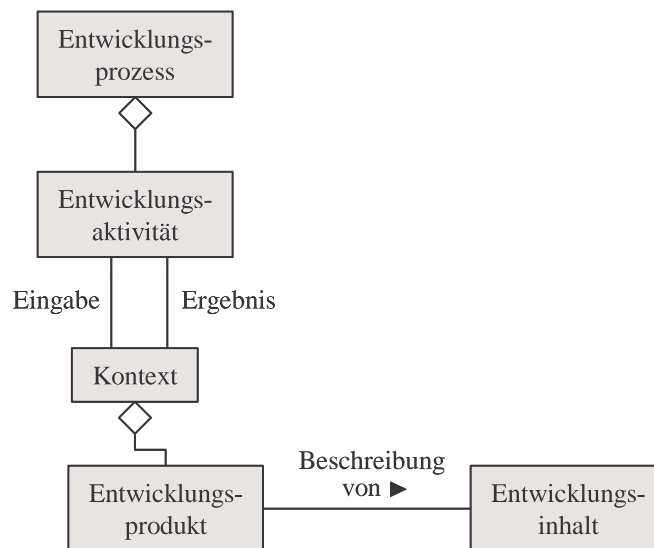


Abbildung 1.1: Elemente von Vorgehensmodellen.

In dieser Arbeit beschäftigen wir uns aus einer Reihe von Gründen mit den drei genannten Entwicklungsaufgaben, die in Abbildung 1.2 hervorgehoben und zusammen mit den weiteren Aufgaben einer Softwareentwicklung dargestellt sind:

- Ein Verständnis der fachlichen Aufgabenstellung, die mit Unterstützung des betrachteten Softwaresystems zu realisieren ist, stellt eine notwendige Voraussetzung für die Entwicklung des Softwaresystems dar. Eine Aufgabenstellung ist umfassend, das heißt nicht auf die Aufgaben des Softwaresystems beschränkt, zu erfassen, da im allgemeinen die fachliche Rolle des Softwaresystems nicht zu Beginn einer Entwicklung feststeht, sondern die Festlegung dieser Rolle eine Entscheidung im Rahmen des Entwicklungsprozesses ist. Die Notwendigkeit einer umfassenden Aufgabenbeschreibung zeigt sich auch am Beispiel von Reengineering-Projekten, in denen eine Anpassung der Rolle des Softwaresystems an eine veränderte Aufgabenstellung das Ziel ist. Je stärker die Verzahnung von Softwarelösungen und ihren Anwendungsbereichen ist, desto wichtiger ist es, die umfassende Aufgabenbeschreibung in die Softwareentwicklung mit einzubeziehen. Auf diese Weise kann die Aufgabenteilung zwischen Softwaresystem und Umgebung dargestellt werden. Für die Entwicklung von Informationssystemen, die typischerweise im Kontext von Unternehmen oder allgemein von Organisationen eingesetzt werden, entspricht die Beschreibung der Aufgabenstellung einer Unternehmensspezifikation, die auf für die Softwareentwicklung relevante Aspekte fokussiert ist.
- Der Anforderungsspezifikation wird in fast allen Vorgehensmodellen große Bedeutung beigemessen. Beispielsweise ist die Durchführung der *Requirements Management* genannten *key process area* des CMM [PCC+93] bereits für die zweite Gütestufe eines Softwareprozesses gefordert. Viele Probleme im Rahmen einer Softwareentwicklung sind, wie beispielsweise in [Bro97a] anhand einer Industriestudie gezeigt,

auf Mängel in der Anforderungsspezifikation zurückzuführen. In nahezu allen Vorgehensmodellen wird das Ziel der Anforderungsspezifikation dadurch charakterisiert, dass geforderte Eigenschaften des zu entwickelnden Softwaresystems aus fachlicher und aus Nutzersicht angegeben werden, was abstrakt zusammengefasst wird in der Forderung, das WAS ohne das WIE zu beschreiben (vergleiche zum Beispiel [DT90], [Bal96], [BDI97]). Dies verdeutlicht den engen Zusammenhang der Anforderungsspezifikation mit der Spezifikation der fachlichen Aufgabenstellung: Wir verstehen die Festlegung der Anteile einer fachlichen Aufgabenstellung, welche durch das zu entwickelnde Softwaresystem zu realisieren sind, als zentralen Bestandteil einer Anforderungsspezifikation.

- Die Anforderungsspezifikation ist das Bindeglied zwischen fachlicher Aufgabenspezifikation und dem Softwaresystem-Entwurf. Im Unterschied zur Anforderungsspezifikation dient der Softwareentwurf der Festlegung des WIE, das heißt dessen, wie das Softwaresystem aus Komponenten aufgebaut ist, und damit der Art und Weise, wie Anforderungen erfüllt werden. Hierbei sind unter anderem die Entscheidungen zu treffen, die für die Erfüllung von Qualitätskriterien, wie Erweiter- und Änderbarkeit, relevant sind.

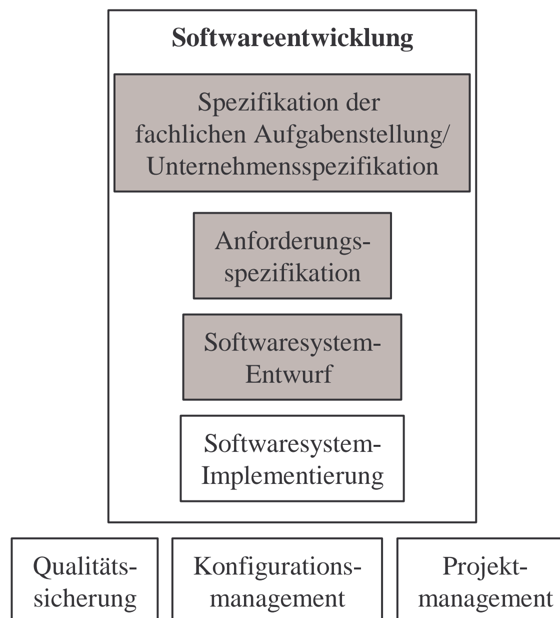


Abbildung 1.2: Softwareentwicklungsaufgaben

Mit der Unternehmens- und Anforderungsspezifikation sowie dem Softwaresystem-Entwurf decken wir die primären Aufgaben einer Softwareentwicklung ab, in denen zentrale Entscheidungen getroffen werden und die somit für den Erfolg einer Entwicklung wesentlich sind. Im allgemeinen wird die Implementierung als eine weitere primäre Aufgabe verstanden, während das Projekt- und Konfigurationsma-

nagement sowie die Qualitätssicherung sekundäre, unterstützende Entwicklungsaufgaben darstellen. Abbildung 1.2 gibt einen Überblick.

Die Spezifikation der fachlichen Aufgabenstellung, die Anforderungsspezifikation und den Softwareentwurf in einem Produktmodell zu behandeln ermöglicht es uns, Zusammenhänge zwischen der Formulierung des WAS und des WIE klar zu erfassen. Dies ist aus Gründen der Verfolgbarkeit von Anforderungen sowie für einen systematischen und schrittweisen Übergang von der Aufgabenstellung zu dessen Realisierung unabdingbar. Wie in Abschnitt 1.1 bereits angedeutet, bleiben in existierenden Vorgehensmodellen gerade diese Zusammenhänge an vielen Stellen unklar. Dies hängt unter anderem damit zusammen, dass in den drei behandelten Entwicklungsaufgaben zwei unterschiedliche Perspektiven einzunehmen sind, die fachliche und die softwaretechnische Perspektive. In der Modellvorstellung, die der fachlichen Perspektive zugrunde zu legen ist, stehen Bausteine fachlicher Aufgaben, wie Objekte und Abläufe der Anwendungswelt, im Vordergrund, während in der softwaretechnischen Modellvorstellung die Bausteine eines Softwaresystems, wie Komponenten und deren Interaktion, zentral sind. Diese Unterschiede führen dazu, dass Zusammenhänge zwischen den Perspektiven und ihren Modellvorstellungen weniger evident sind als zum Beispiel Verfeinerungsbeziehungen innerhalb ein und derselben Modellvorstellung.

Die fachliche Perspektive nehmen wir für die Formulierung der Aufgabenstellung sowie bei der Verteilung von Aufgaben auf Softwaresystem und Umgebung im Rahmen der Anforderungsspezifikation ein. Die softwaretechnische Perspektive ist Grundlage für die Nutzersicht innerhalb der Anforderungsspezifikation sowie für den Softwaresystem-Entwurf. Im Entwurf ist sowohl das Softwaresystem als auch dessen Umgebung zu charakterisieren. Letzteres dient dazu, Umgebungsannahmen explizit zu machen, was gegebenenfalls auch Bestandteil der Anforderungsspezifikation sein kann. Abbildung 1.3 illustriert die Zuordnung der Perspektiven zu Entwicklungsaufgaben.

In der Zielsetzung beschränken wir uns auf die Klasse der verteilten, interaktiven Informationssysteme. Dem liegt folgende Betrachtung zugrunde: Für einen Grossteil der heute zu entwickelnden Softwarelösungen ist Verteilung eine inhärente Eigenschaft (vergleiche etwa [BSI00]). Des weiteren sind Informationssysteme nach wie vor von großer Bedeutung. Zudem handelt es sich bei Informationssystemen häufig um umfangreiche Systeme mit entsprechend hoher Komplexität, so dass sie im Rahmen großer Projekte zu entwickeln sind. Der Einsatz von Vorgehens- und damit von Produktmodellen ist damit von besonderer Bedeutung.

Unter *Informationssystemen* werden im allgemeinen Softwaresysteme verstanden, die der Verwaltung von großen Datenmengen und der Unterstützung komplexer Abläufe dienen (vergleiche beispielsweise [Bre98]). Ein *verteiltes System* wird im allgemeinen als Sammlung von Komponenten (zum Beispiel Prozessen oder Subsystemen) mit einer Möglichkeit zur Kommunikation (zum Beispiel mittels gemeinsamer Speicher oder Nachrichtenaustausch über gemeinsame Kanäle) aufgefasst (vergleiche [Schr93], [BM93]).

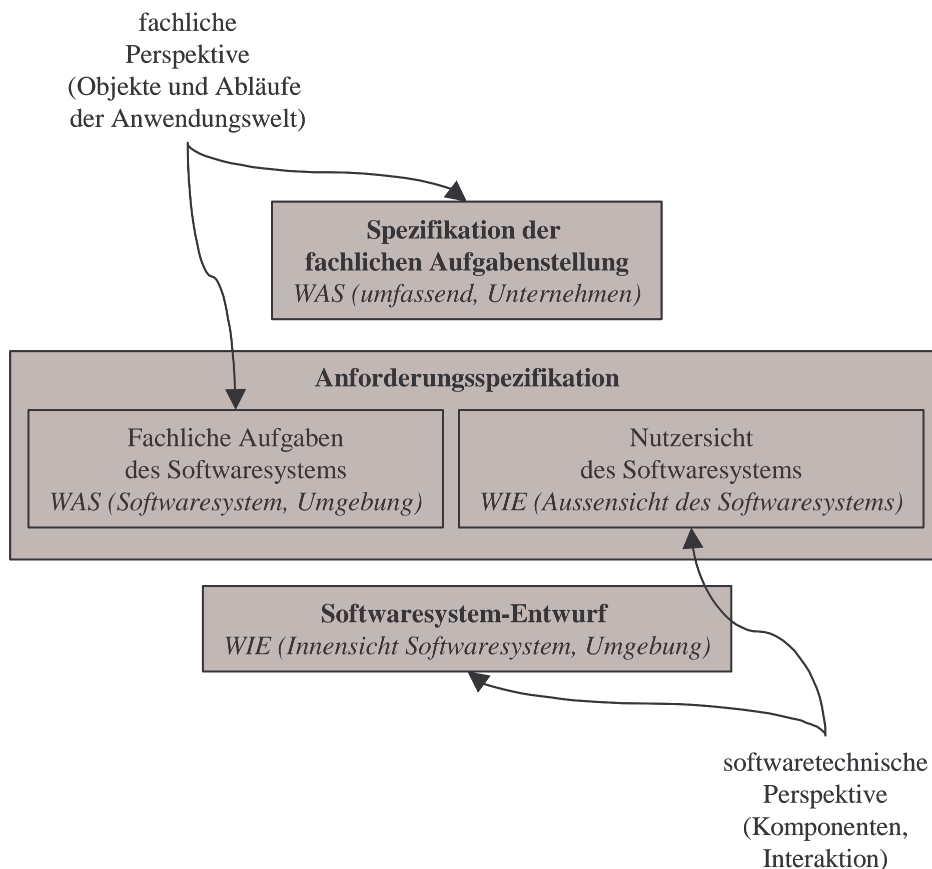


Abbildung 1.3: Entwicklungsaufgaben und Perspektiven.

Neben der Fokussierung auf die drei genannten Entwicklungsaufgaben und auf die Klasse der verteilten Informationssysteme, legen wir in der Zielsetzung die Anwendung mathematischer Modellierungstechniken fest. Mit dem Einsatz mathematischer Techniken können wir deren angesprochene positive Eigenschaften in unser Produktmodell übernehmen. Zu nennen ist zum einen die Präzision der Modelle, was notwendige, wenn auch nicht hinreichende Voraussetzung für ein einheitliches und klares Verständnis von Entwicklungsprodukten ist. Zudem sind methodisch wichtige Eigenschaften, wie Kompositionalität, Kompositionsoperatoren und Verfeinerungsbegriffe, bekannt. Umgekehrt zeigt die Verwendung der Modellierungstechniken zur Definition von Produktarten, für welche Entwicklungsaufgaben diese Techniken sinnvoll einzusetzen sind und wie sie zu einem umfassenden Ansatz kombiniert werden können. Damit leisten wir einen Beitrag, die Lücke zwischen umfassenden Vorgehensmodellen und aufgabenspezifischen formalen Modellierungstechniken der Informatik zu schließen.

1.3 Vorgehensweise

Um unsere Vorgehensweise darzustellen, gehen wir zunächst darauf ein, wie wir anhand eines mathematischen Systemmodells Entwicklungsprodukten eine integrierte Semantik geben. Es folgt ein Überblick der mathematischen Modelle, die wir hierfür einsetzen, sowie der in unserem Produktmodell behandelten Produktarten. Anschließend daran vertiefen wir den in dieser Arbeit zentralen Aspekt der Integration von Entwicklungsinhalten und -produkten.

(System-)Modell der Entwicklungsinhalte.

Da wir durch Produkte Entwicklungsinhalte beschreiben, bildet ein Modell von Entwicklungsinhalten die Grundlage für die Definition von Produktarten. Wir erreichen die Integration der Produktarten unseres Produktmodells, indem wir alle Produktarten anhand eines Modells charakterisieren, das alle Inhalte der drei behandelten Entwicklungsaufgaben integriert. Wir nennen dieses Modell der Inhalte das *Systemmodell*. Abbildung 1.4 stellt den Zusammenhang zwischen Produktarten und Systemmodell schematisch dar. In unserem Systemmodell ist beispielsweise das zu entwickelnde Softwaresystem durch ein Komponentennetzwerk repräsentiert. Anhand dieses Netzwerkes geben wir etwa Topologiespezifikationen und Interaktionsszenarien eine Bedeutung. Während Topologiespezifikationen strukturelle Eigenschaften des Netzwerkes festlegen, adressieren Interaktionsszenarien Verhaltenseigenschaften (vergleiche Abbildung 1.4).

Im einzelnen erfassen wir mit dem Systemmodell folgende Entwicklungsinhalte:

Die Verwaltung fachlicher Daten, durch welche Objekte der Anwendungswelt abgebildet werden, sowie die Unterstützung fachlicher Abläufe sind die typischen Aufgaben von Informationssystemen. Zentrale Entwicklungsinhalte einer fachlichen Aufgabenstellung sind somit fachliche Objekte und entsprechende Datentypen sowie fachliche Abläufe.

Die Rolle, die das zu entwickelnde Softwaresystem innerhalb eines Unternehmens oder einer Organisation einnimmt, ergibt sich daraus, welche fachlichen Objekte und welche Anteile fachlicher Abläufe durch das Softwaresystem zu realisieren sind. Die Aufteilung einer fachlichen Aufgabenstellung auf das zu entwickelnde Softwaresystem und dessen Umgebung stellt daher in unserem Ansatz einen zentralen Inhalt der Anforderungsspezifikation dar.

Aus fachlicher Sicht entspricht das zu entwickelnde Softwaresystem einem Akteur, der fachliche Aktivitäten realisiert und hierzu mit anderen Akteuren interagiert (vergleiche etwa [Pae98b]). Wir modellieren fachliche Akteure ebenso wie die „Bausteine“ eines Softwaresystems durch autonom agierende und miteinander interagierende Komponenten. Damit haben wir zum einen die Möglichkeit, die Nutzungsbeziehung zwischen Umgebung und Softwaresystem in Form von Komponenteninteraktion zu erfassen, was ein weiterer wichtiger Inhalt der Anforderungsspezifikation ist. Zum zweiten können wir auf diese Weise sowohl eine konzeptuelle als auch die reale Dekomposition des Softwaresystems in autonom und nebenläufig agierende Einheiten darstellen, was für die Entwicklung *verteilter* Informationssysteme notwendig ist.

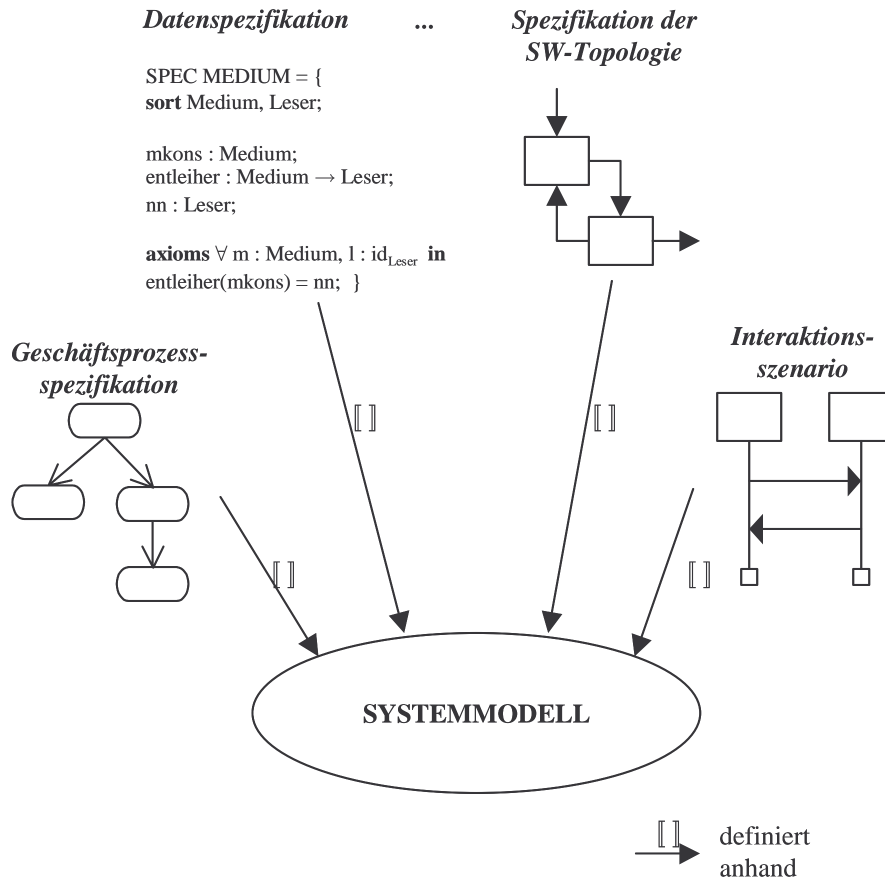


Abbildung 1.4: Definition von Produktarten anhand eines umfassenden Systemmodells.

Verwendete mathematische Modelle.

Wie erwähnt, ist es unser Ziel, Entwicklungsproduktarten präzise und adäquat zu beschreiben, indem wir bewährte mathematische Modelle der Informatik verwenden:

Aus fachlicher Sicht sind wir weniger daran interessiert, Objekte der Anwendungswelt in Form konkreter Datenstrukturen anzugeben. Vielmehr interessieren wir uns für charakteristische Eigenschaften von und Operationen auf fachlichen Objekten. Aus diesem Grund wählen wir in unserem Systemmodell eine algebraische Sicht auf die fachlichen Datensorten und verwenden Σ-Rechenstrukturen, wie sie aus dem Bereich der algebraischen Spezifikation (siehe etwa [Wir90]) bekannt sind, zur Modellierung.

Fachliche Abläufe sind durch die Menge ihrer Ereignisse und deren kausale Abhängigkeiten charakterisiert. Daher lassen sich fachliche Abläufe geeignet durch Modelle abbilden, in denen diese kausalen Abhängigkeiten explizit in Form einer Partialordnung auf der jeweiligen Ereignismenge erfasst werden. *Event structures* [Win87b] und *labelleled partial orders* [Pra86] sind bekannte Modelle dieser Art.

Die Partialität der Kausalordnung lässt es, im Unterschied zu Interleaving-Modellen (vergleiche etwa [Hoa85] und [Lam80]), zu, kausale Unabhängigkeit explizit zu machen. Dadurch können fachliche Abläufe intuitiv modelliert und Überspezifikation vermieden werden. Alle Freiheitsgrade bezüglich der Verteilung und Parallelisierung von Prozessereignissen bleiben auf diese Weise erhalten, was in unserem Ansatz von großer Bedeutung ist, da wir uns auf eine fachliche Aufgabenstellung beziehen, um *Anforderungen* an ein *verteiltes* Informationssystem zu formulieren.

Als Grundlage der komponentenbasierten Sicht des zu entwickelnden Softwaresystems und seiner Umgebung verwenden wir das in FOCUS [BS00] als formale Basis gegebene mathematische Modell. FOCUS ist ein am Lehrstuhl Broy erarbeiteter Rahmen zur Entwicklung verteilter, interaktiver Systeme, in dem ein System als Netzwerk von interagierenden Komponenten modelliert werden kann. Die Komponenten kommunizieren asynchron über gerichtete Kanäle. Komponentenverhalten wird durch stromverarbeitende Funktionen beschrieben, die Eingabehistorien in einen funktionalen Zusammenhang mit (Mengen von) Ausgabehistorien setzen. Dieses Verhaltensmodell baut auf den Kahn-Netzwerken aus [Kah74] auf. Mit FOCUS steht eine breite Palette von Spezifikationsformaten und Beschreibungstechniken ebenso wie Kompositions- und Verfeinerungsbegriffe zur Verfügung. Wie in [Sal02] gezeigt, stellen der in FOCUS gegebene System- und Komponentenbegriff geeignete Abstraktionen aktueller Implementierungstechnologien, wie COM [Box98] und CORBA [Gro98], dar. Um neben Verhaltens- auch topologische Netzwerkeigenschaften festlegen zu können, erweitern wir das mathematische Modell aus FOCUS um Strukturinformation.

Neben der Verhaltenssicht von Komponenten- und Netzwerken als stromverarbeitende Funktionen, modellieren wir Verhalten auch in Form von Spurmengen. Hierbei wählen wir, ähnlich zu [Krü01], Spuren, die sowohl Interaktions- als auch Zustandshistorien umfassen. Dadurch kann ein Bezug zwischen (Daten-)Zustand und Interaktionsverhalten hergestellt werden. Dies ist in unserem Ansatz von Bedeutung, da auf der einen Seite wesentliche Elemente fachlicher Prozesse zustandsbezogene Ereignisse sind, beispielsweise die Änderung des Zustands eines fachlichen Objektes, etwa eines Kontos. Auf der anderen Seite ist das beobachtbare Verhalten von Komponenten auf Interaktion beschränkt.

Um Komponentenverhalten auf konstruktive Weise angeben zu können, setzen wir Automaten ein, die ähnlich zu den in [Rum96] eingeführten Taktautomaten sind. Aufgrund des konstruktiven Charakters eignen sich Automaten für den Feinentwurf von Komponenten und als Implementierungsvorgabe. Die Automatensicht einer Komponente ermöglicht es zudem, Abhängigkeiten zwischen Zustand und Interaktionsverhalten anzugeben.

Die Formulierung fachlicher Anforderungen bedeutet in unserem Systemmodell die Festlegung von Eigenschaften der Realisierungsbeziehung zwischen fachlicher Aufgabenstellung und dem durch Softwaresystem und Umgebung gegebenen Komponentennetzwerk. Im Systemmodell erfassen wir diese Realisierungsbeziehung im wesentlichen durch zwei Abbildungen:

- Durch eine Ähnlichkeitsabbildung zwischen dem fachlichen Zustandsraum und dem durch die Netzwerkkomponenten aufgespannten Zustandsraum stellen wir den Zustandsaspekt einer Realisierungsbeziehung dar. Konkret

verwenden wir das Konzept des Rechenstruktur-Homomorphismus, wie er aus der algebraischen Spezifikation bekannt ist (vergleiche etwa [Wir90]).

- Den Verhaltensaspekt decken wir durch eine Abbildung ab, durch welche Prozessereignissen die zulässigen Realisierungsalternativen in Form einer Menge von Ausführungsfolgen des Komponentennetzwerkes zugeordnet werden. Mittels dieser Abbildung können wir aus der Menge der fachlichen Abläufe, die Menge der zulässigen Ausführungsfolgen des Komponentennetzwerkes ableiten. Dabei stützen wir uns auf die für Partialordnungsmodelle bekannte Step-Semantik ab (vergleiche etwa [GG98]).

Produktmodell.

Bei der Definition der verschiedenen Arten von Entwicklungsprodukten verfolgen wir einen modularen Ansatz. Um die Ergebnisse „kleiner“ Entwicklungsschritte festhalten zu können, definieren wir Produktarten, die sich auf einen bestimmten Entwicklungsinhalt beschränken, zum Beispiel die Spezifikation eines abstrakten Datentypen. Die Gesamtheit dieser grundlegenden Produktarten deckt alle methodisch wichtigen Sichtweisen auf jeden einzelnen Inhalt unseres Systemmodells ab. Wir verwenden grundlegende Produktarten als Bausteine für umfassendere, im allgemeinen mehrere Inhalte adressierende Produktarten. Mit Hilfe umfassender Produktarten strukturieren wir Entwicklungsergebnisse im Grossen, die typischerweise nicht Ergebnis einzelner Entwicklungsaktivitäten, sondern von Phasen darstellen. Ein Beispiel ist die Produktart der Anforderungsspezifikation als Ergebnis der Anforderungsanalyse-Phase.

Grundsätzlich differenzieren wir Produktarten sehr fein, um den jeweiligen methodischen Zweck einer Produktart klar zum Ausdruck bringen zu können. So unterscheiden wir beispielsweise zwischen der exakten und der exemplarischen Spezifikation eines Geschäftsprozesses.

Die Elemente unseres Produktmodells lassen sich wie folgt grob charakterisieren:

Die Produktarten, die wir zur Beschreibung fachlicher Datensorten einführen, orientieren sich an Techniken der algebraischen Spezifikation.

Die Spezifikation fachlicher Abläufe strukturieren wir durch Produktarten zur Angabe von Lebendigkeits- und Sicherheitseigenschaften, was für die Verhaltensspezifikationen verteilter Systeme angemessen und bewährt ist. Wir zeigen, dass die aus der Praxis bekannten Geschäftsprozessbegriffe eine spezielle Form von Lebendigkeitseigenschaft darstellen.

Die Spezifikation des zu entwickelnden Softwaresystems und dessen Umgebung bedeutet die Angabe von topologischen und von Verhaltenseigenschaften der jeweiligen Komponenten-Netzwerke. Die getrennte Spezifikation unterschiedlicher Aspekte unterstützen wir durch Produktarten, mit Hilfe derer die topologischen Aspekte von Netzwerken unabhängig von Verhaltensaspekten beschrieben werden können. Die Verhaltensspezifikation von Netzwerken decken wir, wie für fachliche Abläufe, durch Produktarten zur Spezifikation von Lebendigkeits- und Sicherheitseigenschaften ab. Ergänzt werden diese durch Produktarten zur Charakterisierung der Automatensicht und des beobachtbaren Verhaltens einzelner Komponenten.

1.3.1 Integration

Aufgrund des Ziels, in Form unseres Produktmodells eine Grundlage für umfassende und durchgängige Entwicklungsprozesse zu schaffen, ist die *Integration* einer Vielzahl von Entwicklungsinhalten und -Produkten ein wesentlicher Aspekt dieser Arbeit. Unter Integration wird im allgemeinen der Zusammenschluss sowie die Verbindung zu einem Ganzen verstanden. Konkret steht der Begriff der Integration beispielsweise in der Sprachwissenschaft für die Verschmelzung unterschiedlicher Sprachen zu einer Schriftsprache. In dieser Arbeit findet Integration an verschiedenen Stellen statt, auf die wir im Folgenden eingehen.

Integration von Entwicklungsinhalten im Systemmodell.

Grundlegend für die Integration von Entwicklungsprodukten ist die Integration ihrer Inhalte. Wir erreichen die Integration der Inhalte der drei behandelten Entwicklungsaufgaben, indem wir die Modelle der verschiedenen Inhalte in unserem Systemmodell zueinander in Bezug setzen.

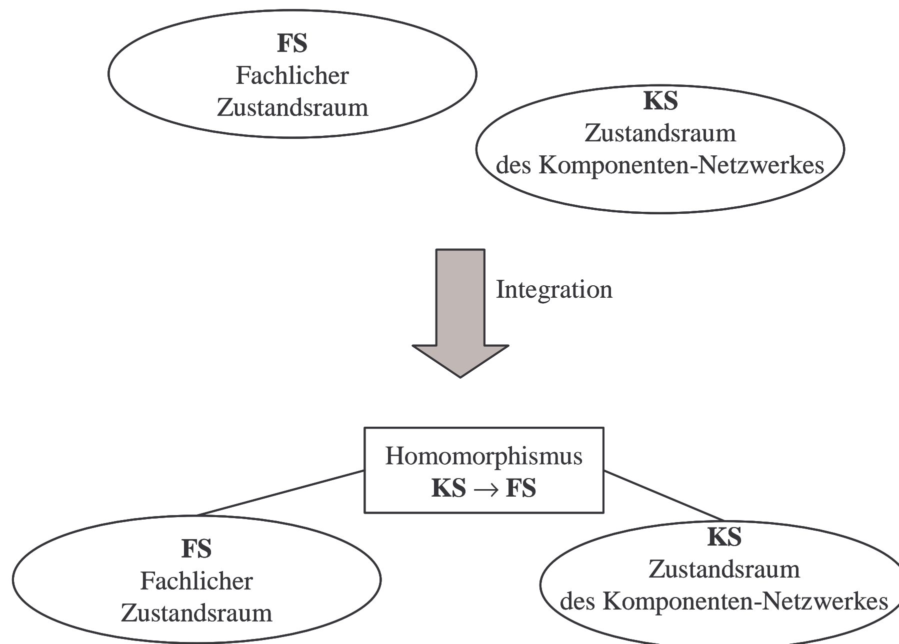


Abbildung 1.5: Integration von Zustandsräumen im Systemmodell.

Beispiel eines Entwicklungsinhaltes ist der für eine Software-Entwicklung relevante Ausschnitt des Zustandsraums eines Unternehmens. Einen weiteren Entwicklungsinhalt stellt der Zustandsraum dar, der durch die Komponenten des zu entwickelnden Softwaresystems und dessen Umgebung gebildet wird. Letzteren bezeichnen wir im Folgenden als Netzwerk-Zustandsraum. Beide Zustandsräume modellieren wir durch Rechenstrukturen, welche jeweils die ausgezeichnete, mit *State* bezeichnete Sorte der Zustandswerte umfassen. Von einer sinnvollen Entwicklung erwarten wir, dass der fachliche durch den Netzwerk-Zustandsraum reali-

siert wird. Das heißt, dass fachliche Information durch die Komponenten des Software-Umgebungs-Netzwerkes geeignet abgebildet wird. Dementsprechend integrieren wir beide Entwicklungsinhalte im Systemmodell anhand einer Ähnlichkeitsabbildung zwischen den Zustandsräumen (vergleiche Abbildung 1.5). Konkret ist die Ähnlichkeitsabbildung ein Rechenstruktur-Homomorphismus. Selbst wenn diese Ähnlichkeitsabbildung im Rahmen einer Entwicklung nicht explizit angegeben wird, so haben wir damit im Systemmodell in jedem Fall die Forderung der Existenz einer solchen Abbildung aufgestellt und damit ein Konsistenzkriterium für die beiden genannten Entwicklungsinhalte formuliert.

Integration von Produktarten.

Wie eingangs des Kapitels erwähnt, entstehen im Rahmen einer schrittweisen Entwicklung eine Vielzahl unterschiedlicher Arten von Entwicklungsprodukten. Daraus ergibt sich die Notwendigkeit der Integration der verschiedenen Produktarten, das heißt deren „Verschmelzen zu einer Spezifikationssprache". Wir erreichen diese Integration, indem wir die Bedeutung *aller* Produktarten unseres Produktmodells anhand des *einen* Systemmodells definieren (vergleiche Abbildung 1.4). Auf diese Weise ergeben sich Zusammenhänge zwischen Produkten aus den im Systemmodell festgelegten Zusammenhängen zwischen Entwicklungsinhalten. Wir sprechen von einer integrierten Produktsemantik.

Beispielsweise sind zwei Produkte, die den fachlichen beziehungsweise den Netzwerk-Zustandsraum eindeutig festlegen nur dann konsistent, wenn zwischen den beiden spezifizierten Zustandsräumen eine dem Systemmodell entsprechende Ähnlichkeitsabbildung existiert. Legt eines der Produkte nur einige Eigenschaften des Netzwerk-Zustandsraums fest, dann ist es mit dem anderen Produkt vereinbar, falls ein Netzwerkzustandsraum existiert, der die geforderten Eigenschaften erfüllt und zudem (gemäß dem Systemmodell) ähnlich zu dem eindeutig spezifizierten fachlichen Zustandsraum ist.

Partialität von Produkten.

Eine weitere Anforderung an eine integrierte (Produkt-)Semantik ergibt sich aus der Tatsache, dass wir mit einem Produkt im allgemeinen nicht die Gesamtheit der Inhalte einer Entwicklung, sondern nur einen Teil davon erfassen, da

- nur bestimmte Aspekte, wie etwa Struktur ohne Verhalten
- nur bestimmte „Bausteine", wie etwa nur eine bestimmte Komponente des betrachteten Softwaresystems oder nur ein bestimmter Geschäftsprozess
- nur bestimmte, aber nicht notwendigerweise alle Eigenschaften von Inhalten, etwa nur Sicherheits- und keine Lebendigkeitseigenschaften des Verhaltens oder nur einige aber nicht alle Lebendigkeitseigenschaften

beschrieben werden. Durch ein Produkt legen wir also im allgemeinen einige Eigenschaften bestimmter Inhalte fest und lassen andere Eigenschaften der adressierten Inhalte sowie die Eigenschaften der anderen Inhalte einer Entwicklung offen. Ein Produkt kann daher als partielle Spezifikation verstanden werden. Diese Partia-

lität muss sich auf geeignete Weise in der Bedeutung, das heißt der Semantik von Produkten widerspiegeln. Wir interpretieren daher Produkte wie folgt:

> *Jede Gesamtheit von Entwicklungsinhalten erfüllt die durch ein Produkt gegebene Spezifikation, wenn sie die für die adressierten Inhalte formulierten Eigenschaften erfüllt. Alle offen gelassenen Eigenschaften können beliebig ausgeprägt sein, solange die im Systemmodell definierten Zusammenhänge zwischen Inhalten erfüllt sind.*
>
> *Wir bezeichnen eine Gesamtheit von Entwicklungsinhalten, die eine Ausprägung des Systemmodells darstellt, auch als ein* Entwicklungssystem. *Ein Produkt steht damit für die Menge von Entwicklungssystemen, welche das Produkt (entsprechend obiger Festlegung) erfüllen.*

Diese Form der Semantik, die ähnlich zu der sogenannten losen Semantik aus Ansätzen algebraischer Spezifikationen ist und einer Interpretation von Produkten im Sinne der Unterspezifikation entspricht, ist in Abbildung 1.6 schematisch dargestellt.

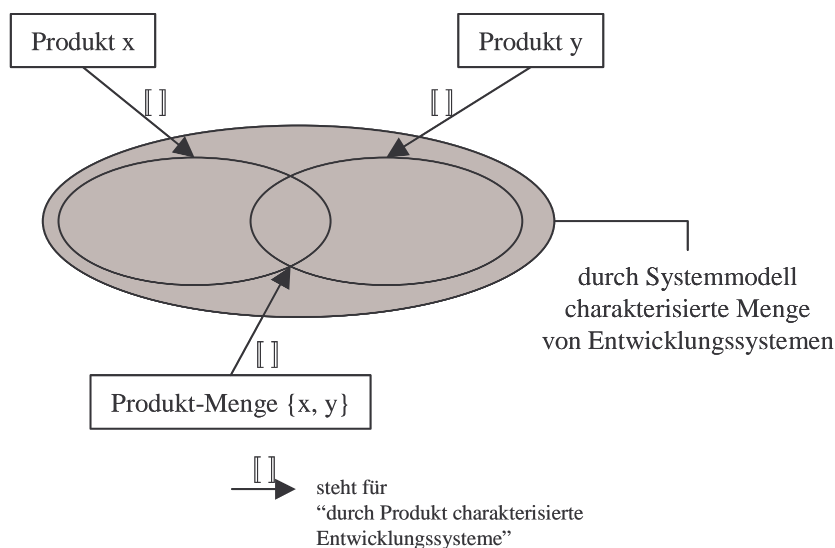


Abbildung 1.6: Semantik von Produkten und Produktmengen.

Das folgende Beispiel illustriert die oben beschriebene Produktsemantik: Sei ein Produkt gegeben, welches ausschließlich die Kommunikationsverbindungen, das heißt die syntaktische Schnittstelle, einer Komponente k des zu entwickelnden Softwaresystems festlegt. Das Produkt wird durch jedes Entwicklungssystem erfüllt, das die Komponente k mit entsprechender Schnittstelle als Teil des Softwaresystems enthält. Ein das Produkt erfüllendes Entwicklungssystem kann beispielsweise beliebige weitere Komponenten mit beliebigen Schnittstellen umfassen solange dadurch die im Systemmodell für ein Netzwerk geforderten Eigenschaften erhalten bleiben. In unserem Systemmodell lassen wir beispielsweise nur Punkt-zu-Punkt-Verbindungen zwischen Komponenten zu. Da mit der Schnittstellenspezifikation der Komponente k weder explizit noch implizit Aussagen über Zustandseigenschaften gemacht werden, kann zudem beispielsweise auch der fachliche Zu-

standsraum eines Entwicklungssystems, ebenso wie andere nicht adressierte Inhalte, beliebig ausgeprägt sein.

Die gewählte Produktsemantik hat den methodisch wichtigen Vorteil, dass wir Entwicklungsprodukte auf sehr einfache und intuitive Weise zu umfassenden Produkten einer Entwicklung komponieren und damit integrieren können: Aus logischer Sicht können wir Komposition durch Konjunktion abbilden, aus mengentheoretischer Sicht ergibt sich die Bedeutung einer Menge von Produkten durch Bildung der Schnittmenge der Entwicklungssystem-Mengen, die durch jedes einzelne Produkt definiert sind (vergleiche Abbildung 1.6). Dadurch wird insbesondere die Integration aller im Rahmen einer Entwicklung erstellten Produkte möglich. Zudem ergibt sich auf diese Weise ein klares Konsistenzkriterium: eine Produktmenge ist genau dann konsistent, wenn sie eine nichtleere Menge von Entwicklungssystemen beschreibt; das heißt, wenn mindestens ein Entwicklungssystem existiert, das die durch die Produktmenge geforderten Eigenschaften erfüllt.

1.4 Ergebnisse

Mit dem in dieser Arbeit vorgestellten Produktmodell schaffen wir eine Grundlage für die Definition von Entwicklungsprozessen für verteilte Informationssysteme. Unser Produktmodell deckt drei Kernaufgaben einer Systementwicklung ab, wobei die vorgestellten Produktarten so aufeinander abgestimmt sind, dass bruchfreie Übergänge von der Unternehmensspezifikation, über die Anforderungsspezifikation bis hin zum detaillierten Softwaresystem-Entwurf ermöglicht werden.

Mit der feinen Differenzierung von Produktarten, etwa zwischen partiellen und exakten Datentypspezifikationen, stellen wir Elemente zur Dokumentation von Entwicklungsergebnissen im Kleinen wie im Grossen zur Verfügung, was für ein schrittweises Vorgehen ebenso von Bedeutung ist, wie etwa für ein effektives Änderungsmanagement. Eine solche Differenzierung ist aus einigen formalen Methoden bekannt, in existierenden Vorgehensmodellen aber nicht oder nur in Ansätzen zu finden.

Das in dieser Arbeit entwickelte Systemmodell ist zum einen als semantische Basis des Produktmodells von Bedeutung. Darüber hinaus geben wir mit unserem Systemmodell eine umfassende, mathematisch fundierte und damit präzise Begriffswelt, das heißt Ontologie, für Kerninhalte einer Informationssystementwicklung an. Unsere Arbeit gibt somit ein Beispiel dafür, wie ein Schwachpunkt vieler existierender Vorgehensmodelle, nämlich das Fehlen eines expliziten Modells der Entwicklungsinhalte, überwunden werden kann. Explizit und präzise formulierte Entwicklungsinhalte helfen, ein einheitliches Verständnis der an einer Entwicklung Beteiligten zu erreichen und somit Missverständnisse zu vermeiden, was wiederum Voraussetzung für eine effiziente Entwicklung ist.

Im Einzelnen erzielen wir mit dem Systemmodell folgende Ergebnisse:

Als Grundlage der Unternehmensspezifikation umfasst das Systemmodell ein Modell fachlicher, abstrakter Datentypen und ein Modell des aus zustandskapselnden Objekten modular aufgebauten, fachlichen Zustandsraums. Wir integrieren beide

Modelle, indem wir Objekte anhand der abstrakten Datentypen klassifizieren und damit festlegen, welche Zustände ein Objekt einnehmen kann und welche Zugriffsoperationen auf Objektzuständen zulässig sind. Damit schaffen wir die Möglichkeit, wertbasierte algebraische Techniken im Kontext einer zustandsbasierten Modellierung einzusetzen. Dies ermöglicht beispielsweise die Kombination algebraischer Techniken mit der Entity-Relationship-Modellierung [Che76] oder mit objektbasierten Ansätzen (siehe etwa [CY91], [JEJ94]).

Unser Daten- und Zustandsmodell garantiert zudem klare Spezifikationen unter anderem dadurch, dass die Seiteneffektfreiheit von Operationen auf zustandskapselnden Einheiten garantiert ist. Dies ist ein signifikanter Vorteil gegenüber objektorientierten Modellen, in denen Objekte als interagierende Einheiten verstanden werden. Gegenüber der Entity-Relationship-Modellierung hat unser Ansatz einen Vorteil mit objektorientierten Modellen gemeinsam. Neben Selektoroperationen, das heißt Attributen, können auch die zulässigen zustandsändernden Operationen durch Klassifikation von Objekten, beziehungsweise von Entitäten, festgelegt werden.

Ein praxistaugliches Modell fachlicher Abläufe erhalten wir, indem wir diese durch einen Partialordnungsansatz erfassen, wodurch der konzeptuelle Kern bewährter Techniken zur Geschäftsprozessspezifikation, wie Petri-Netze [Rei95] und ereignisgesteuerte Prozessketten [Schee98], abgedeckt ist. Geschäftsprozessmodelle sind zudem geprägt durch zustandsbezogene Prozessereignisse: die Abfragen von Zustandseigenschaften bestimmen den Verlauf von Prozessen und Zustände werden in Prozessschritten verändert. Daher sind Zustands- und Verhaltensmodell in Bezug zu setzen. Wir erreichen dies in dieser Arbeit dadurch, dass wir die Anwendung der, wie oben beschrieben, durch Klassifikation festgelegten Zugriffsoperationen auf fachliche Objekte, als Teilmenge der zulässigen Prozessereignisse festlegen. Damit integrieren wir fachliche Daten-, Zustands- und Verhaltenssicht zu einem umfassenden Modell fachlicher Aufgabenstellungen von Informationssystemen.

Neben der Aufgabenstellung, ist die softwaretechnische Sicht seiner Realisierung Inhalt des Systemmodells. Wie erwähnt, modellieren wir das Softwaresystem und seine Umgebung als Komponentennetzwerk und stützen uns dabei auf das Verhaltensmodell der stromverarbeitenden Funktionen der Methode FOCUS [BS00]. FOCUS konzentriert sich auf die Verhaltensspezifikation. Da ein Softwaresystem neben dem Verhalten auch durch seine Topologie, das heißt durch die Dekomposition in Komponenten, charakterisiert ist, stellt die Spezifikation topologischer Eigenschaften einen wichtigen Teil eines Softwareentwurfs dar. Beispielsweise kann die Topologie eines Softwaresystems entscheidend für die Erfüllung von Qualitätskriterien, wie Erweiterbarkeit, sein. Wir ermöglichen mit unserem Systemmodell die integrierte Spezifikation von Topologie und Verhalten, indem wir das Verhaltensmodell aus FOCUS um strukturelle Aspekte von Netzwerken ergänzen.

In Entwicklungsstandards, wie beispielsweise dem V-Modell [BDI97] und der IEEE Software Requirements Specification (SRS) [DT90], wird gefordert, dass im Rahmen einer Anforderungsspezifikation Anforderungen an das Softwaresystem aus *fachlicher* Sicht gegeben werden, wobei diese Sichtweise abstrakt dadurch charakterisiert wird, dass das WAS ohne das WIE zu beschreiben ist. Was dies konkret bedeutet, bleibt teilweise unklar. Im Unterschied dazu, treffen wir in unse-

rem Systemmodell eine klare Unterscheidung zwischen dem WAS, repräsentiert durch die fachliche Aufgabenstellung, und dem WIE, repräsentiert durch das Komponentennetzwerk von Softwaresystem und Umgebung. Zusätzlich geben wir im Systemmodell Elemente zur Charakterisierung der Realisierungsbeziehung zwischen Aufgabenstellung und Softwaresystem-Umgebungs-Netzwerk an, womit uns ein klarer Begriff fachlicher Anforderungen, als Eigenschaften dieser Realisierungsbeziehung, zur Verfügung steht.

Wesentliches Element der Realisierungsbeziehung ist die Interpretation eines fachlichen Ablaufs als Menge von Ausführungsspuren eines Komponentennetzwerkes. Hierzu führen wir die bekannte Step-Semantik von Partialordnungsmodellen (vergleiche etwa [GG98]) weiter, zu einer Semantik die auf die Spezifika unseres Prozess- und Spurmodells eingeht. Damit stellen wir einen Bezug her zwischen einem Partialordnungsmodell mit echter Nebenläufigkeit und einem Spurmodell, in dem Parallelität durch Interaktion auf parallelen Kanälen modelliert wird. Zudem wird der in beiden Modellen enthaltene Zustandsaspekt geeignet berücksichtigt. Damit setzen wir ein für die fachliche Verhaltenssicht geeignetes Modell zu einem für die softwaretechnische Verhaltenssicht geeigneten Modell in Bezug, und leisten einen wichtigen Beitrag zur Integration von Unternehmens- und Softwaremodellierung.

Die mit dem in dieser Arbeit definierten Produktmodell erreichten Ergebnisse fassen wir wie folgt zusammen:

Die Produktarten zur fachlichen Datenspezifikation wählen wir geeignet für den Einsatz algebraischer Techniken, jedoch unabhängig von spezifischen Notationen. Für eine schrittweise Entwicklung ist es nötig, zwischen Spezifikationen unterscheiden zu können, die einige aber nicht notwendigerweise alle Eigenschaften adressierter Datensorten und Operationen angeben, und solchen, mit denen wir den Anspruch verbinden, alle Eigenschaften anzugeben. Hierzu führen wir entsprechend differenzierte Arten von Entwicklungsprodukten ein, was als Verallgemeinerung des bekannten Konzepts hierarchischer Spezifikationen (vergleiche etwa [WPP+83] und [Bre91]) verstanden werden kann und wodurch wir den systematischen Einsatz algebraischer Spezifikationstechniken weitergehend unterstützen.

Die Strukturierung von Verhaltensspezifikationen reaktiver Systeme in Lebendigkeits- und Sicherheitseigenschaften ist bekannt und bewährt, so dass wir uns bei der Definition der Produktarten zur (fachlichen) Verhaltensspezifikation daran orientieren. Neu ist jedoch eine Form von Lebendigkeitseigenschaft, die eine Verallgemeinerung der, aus dem Bereich der temporalen Logik (vergleiche zum Beispiel [MP92]) bekannten Klasse der Reaktionseigenschaften (*leadsto* oder *response properties*) darstellt. Wir analysieren praxisrelevante Definitionen des Geschäftsprozessbegriffes und zeigen, dass dieser Reaktionseigenschaften entspricht, in denen Reaktionen nicht, wie bisher üblich, durch atomare Ereignisse, sondern durch (im allgemeinen aus mehreren, kausal geordneten Ereignissen bestehende) Prozesse beschrieben werden. Durch diese Verallgemeinerung kommen wir zu einer praxistauglichen, gut skalierenden Verhaltenssicht. Wir geben eine mathematische Definition dieser Form von Verhaltenseigenschaft und damit des Geschäftsprozessbegriffes an. In Form fein differenzierter Produktarten, wie Geschäftsprozessszenarien und exakten Geschäftsprozessspezifikationen, stellen wir Mittel zur schrittweisen Charakterisierung von Geschäftsprozessen zur Verfügung.

Die Spezifikation von Eigenschaften der Außensicht (Black Box View) verteilter Systeme in Form von Komponentennetzwerken anzugeben, ist bewährt und entspricht der Anwendung des Prinzips „divide et impera“. Die Methode FOCUS [BS00] führt diese Form der Spezifikation unter dem Begriff der zusammengesetzten Spezifikationen (*composite specifications*) ein. Wie erwähnt, ist jedoch auch die Spezifikation der Innensicht (Glass Box View) und damit der Topologie eines verteilten Systems, eine bedeutsame Entwicklungsaufgabe. Wir führen daher Produktarten zur Spezifikation der Innensicht ein, die eine Unterscheidung erlauben, zwischen der Festlegung von in einer Topologie einzuhaltenden Schnittstellen und der exakten Festlegung einer Topologie. Damit fassen wir in existierenden Ansätzen nur unscharf differenzierte Produktarten, die häufig als logische oder konzeptuelle sowie als technische Architektur bezeichnet werden, präzise.

Neben Architektur-Spezifikationen steht die Produktart der Anforderungsspezifikation für ein Entwicklungsergebnis im Grossen. Der Zweck einer Anforderungsspezifikation ist die Charakterisierung des zu entwickelnden Softwaresystems aus fachlicher und aus Nutzersicht. Diese Sichtweisen können wir anhand der in unserem Systemmodell formulierten Realisierungsbeziehung, zwischen der Aufgabenstellung und dem, aus dem Softwaresystem und dessen Umgebung gebildeten Komponentennetzwerk, klar fassen. Dies ermöglicht uns die präzise Konkretisierung des Kerns von in Entwicklungsstandards vorgegebenen Produktarten, wie dem sogenannten *Pflichtenheft* [Bal96], den *Anwenderforderungen* des V-Modells [BDI97], oder der *IEEE Software Requirements Specification* aus [DT90].

1.5 Verwandte Arbeiten

Die vorliegende Arbeit ist durch eine Vielzahl existierender Arbeiten beeinflusst, auf die wir im Folgenden eingehen.

Erste Ansätze zur Erzeugung eines Synergieeffektes zwischen pragmatischen und formalen Methoden werden in den Arbeiten von Hußmann [Huß93a, Huß93b] deutlich, der diese Idee in [Huß97] mit den in SSADM verwendeten Beschreibungstechniken ausgeführt hat.

Die Vorgehensweise, anhand eines umfassenden mathematischen Systemmodells unterschiedlichen spezifikatorischen Sichten eine präzise und integrierte Semantik zu geben, übernehmen wir aus [Rum96]. In [Rum96] wird ein mathematisches Modell für den Entwurf verteilter objektorientierter Systeme definiert. Ähnlich zur vorliegenden Arbeit dient dabei das aus der Methode FOCUS [BS00] bekannte Komponentenmodell als Grundlage. Ein Unterschied besteht darin, dass wir Komponenten in ihrer in FOCUS vorgestellten Allgemeinheit verwenden, während in [Rum96] spezielle Arten von Komponenten definiert werden, die eine geeignete Modellierung von Objekten im Sinne der Objektorientierung darstellen.

In [Rum96] stehen Entwicklungsproduktarten, die dort Dokumentarten genannt werden, im Vordergrund, die für den Entwurf objektorientierter Softwaresysteme bestimmt sind, so dass diese Dokumentarten als Spezialisierungen beziehungsweise Ergänzungen von Produktarten verstanden werden können, wie wir sie in Kapitel 5 zur komponentenbasierten Spezifikation einführen. Insbesondere wird in

[Rum96] ein Automatenmodell definiert, das wir auch in unserer Arbeit einsetzen. Der in [Rum96] vorgestellte Verfeinerungskalkül für Automaten ist somit auch im Kontext unseres Produktmodells anwendbar.

Auch [Bro95] ist eine der ersten Arbeiten, in denen mehrere mathematische Modelle zu einem umfassenden Systemmodell kombiniert werden, um darauf aufbauend unterschiedlichen spezifikatorischen Sichten eine integrierte Semantik zu geben. Das in [Bro95] vorgestellte Systemmodell enthält Elemente der komponentenbasierten Perspektive unseres Systemmodells auf hohem Abstraktionsniveau.

Ähnlich zur vorliegenden Arbeit, ist es ein Ziel von [Pae98b] einen Bezug zwischen der Unternehmensspezifikation und der Softwaresystemspezifikation herzustellen. Im Unterschied zu unserer Arbeit werden in [Pae98b] Entwicklungsinhalte und -produktarten im wesentlichen informell beschrieben, da der Fokus nicht auf Details der Daten-, Zustands- und Verhaltensmodellierung liegt, sondern vielmehr auf einer methodischen Vorgehensweise, um systematisch von einer Unternehmensspezifikation, über die Spezifikation des Softwaresystems aus Nutzersicht, zu einem objektorientierten Entwurf des Softwaresystems zu kommen. Daher kann den in [Pae98b] eingesetzten Modellierungskonzepten, erfasst in dem sogenannten Konzeptmodell, anhand unseres Systemmodells eine mathematische Bedeutung gegeben werden und auf diese Weise eine Festlegung von offen gelassenen Details getroffen werden. Ein ähnlicher Zusammenhang besteht zu den Produktarten aus [Pae98b]. Somit stellt die in [Pae98b] vorgeschlagene Vorgehensweise einen möglichen, geeigneten methodischen Rahmen für den Einsatz unseres Produktmodells.

Wie in der vorliegenden Arbeit, werden in [Web91] Anforderungen an ein Softwaresystem bezüglich einer globalen, das heißt umfassenden und ganzheitlich beschriebenen Aufgabenstellung formuliert und in einem komponentenbasierten Entwurf des Softwaresystems umgesetzt. Sowohl Entwicklungsinhalte als auch spezifikatorische Sichten werden in [Web91] mathematisch definiert. Im Unterschied zur vorliegenden Arbeit, werden in [Web91] Daten- und Datenzustandsaspekte nicht explizit modelliert, was jedoch für die Entwicklung von Informationssystemen wesentliche Elemente sind. Ähnlichkeiten bestehen bei der Verhaltensspezifikation in Form von Sicherheits- und Lebendigkeitseigenschaften. Für den Übergang von globalen zu komponentenlokalen Verhaltenseigenschaften wird in [Web91] eine detaillierte methodische Vorgehensweise vorgeschlagen, die auch für den Einsatz unseres Produktmodells wertvolle Hilfestellung bieten kann.

Während in [Web91] die globale Verhaltenssicht auf einem Interleaving-Modell basiert, wird mit [Bro89] ein erster Ansatz unternommen Elemente der Methodik aus [Web91] auf ein Partialordnungsmodell, ähnlich zu unserem Modell fachlicher Abläufe, zu übertragen. Damit liegt [Bro89] in diesem Punkt näher an unserem Ansatz als [Web91].

In Abschnitt 1.3 erwähnen wir den Einsatz eines auf Spurmengen basierenden Verhaltensmodells, ähnlich dem in [Krü01] vorgestellten. Der in [Krü01] behandelte, methodische Einsatz von Sequenzdiagrammen bietet daher wertvolle Hilfestellung für die Verwendung dieser Notation für unsere Produktarten zur Spezifikation von Netzwerkverhalten. Auch der Übergang von komponentenübergreifenden Verhalteneigenschaften zu komponentenlokalen Automatenspezifikationen wird in [Krü01] behandelt.

In [Bre91] wird ein Rahmenwerk zur Integration zustandsfreier, algebraischer Spezifikation abstrakter Datentypen mit der zustandsbehafteten Beschreibung von Klassen im Sinne imperativer objektorientierter Programmiersprachen definiert. Hierfür wird, wie in unserem Systemmodell, die abstrakte zustandsfreie Sicht mit der konkreten zustandsbehafteten Sicht in Bezug gesetzt. Jedoch orientiert sich die zustandsbasierte Sicht in [Bre91] an Objektgeflechten, wie sie durch objektorientierte Programmiersprachen definiert werden können. Im Unterschied dazu, wählen wir in der vorliegenden Arbeit ein einfacheres, für unseren Zweck der fachlichen Zustandsmodellierung passenderes Zustandsmodell, das auf einfachen Belegungsabbildungen basiert. Ein weiterer Unterschied besteht darin, dass in [Bre91] die beiden Sichten in Form einer Korrektheitsbeziehung korreliert werden, während in unserem Ansatz durch abstrakte Datentypen Eigenschaften der zustandsbasierten Sicht determiniert sind.

Viel Einfluss auf die in der vorliegenden Arbeit definierten Produktarten haben eine Reihe praxistauglicher, informeller Vorgehensmodelle, die wir bereits in Abschnitt 1.3 nennen und die von allgemeinen Standards, wie dem deutschen V-Modell [BDI97], bis hin zu spezifischeren Ansätzen, wie dem Unified Process [JBR99], reichen.

Für die Entwicklungsprodukte zur Formulierung einer fachlichen Aufgabenstellung orientieren wir uns an Ansätzen der Unternehmensspezifikation, wie [Schee98], [Öst95a], [WFMC99] und [OHM+91] und der oben bereits genannten Arbeit von Paech [Pae98b].

1.6 Aufbau der Arbeit

Aus der oben geschilderten Vorgehensweise ergibt sich folgender Aufbau der Arbeit:

In Kapitel 2 gehen wir zunächst darauf ein, was wir unter einem Systemmodell der Entwicklungsinhalte und unter Entwicklungsprodukten verstehen, und auf welche Weise wir die Semantik von Entwicklungsprodukten anhand des Systemmodells angeben. Anschließend daran definieren wir das Systemmodell in folgender Reihenfolge:

- Ganzheitliche Perspektive: Systemmodell-Anteil zur Modellierung einer fachlichen Aufgabenstellung
- Komponentenbasierte Perspektive: Anteil zur Modellierung des zu entwickelnden Softwaresystems und seiner Umgebung
- Perspektiven-Zusammenhänge: Anteil zur Modellierung der Realisierungsbeziehung zwischen Aufgabenstellung und dem betrachteten Softwaresystem zusammen mit dessen Umgebung

In Kapitel 3 bis 5 definieren wir grundlegende Produktarten, die im wesentlichen einen Systemaspekt adressieren. Die in Kapitel 3 definierten Produktarten dienen der Spezifikation fachlicher Datentypen und des fachlichen Zustandsraums. Die Spezifikation fachlicher Abläufe behandeln wir in Kapitel 4. Produktarten zur

komponentenbasierten Spezifikation von Softwaresystem und Umgebung sind Gegenstand von Kapitel 5.

Unter Verwendung der grundlegenden Produktarten geben wir in Kapitel 6 komponierte Produktarten zur Erfassung umfassender Entwicklungsergebnisse an, darunter die Produktart der Anforderungsspezifikation sowie der Schnittstellen- und Realisierungsarchitektur des Softwaresystems.

In Kapitel 7 gehen wir auf Themen ein, die im Umfeld dieser Arbeit liegen.

Kapitel 8 enthält Definitionen grundlegender Begriffe und Konzepte, die wir in unserer Arbeit verwenden, dabei aber im wesentlichen auf existierende Arbeiten zurückgreifen.

1.7 Durchgängiges Beispiel

Die eingeführten Entwicklungsinhalte und Entwicklungsproduktarten verdeutlichen wir durchgehend anhand der Entwicklung eines Softwaresystems zur Verwaltung einer Bibliothek. Hierbei beschränken wir uns auf einen Ausschnitt der Gesamtfunktionalität und nehmen Vereinfachungen vor, um den Umfang gering zu halten und trotzdem möglichst viele Gesichtspunkte anzusprechen.

Wir nehmen an, dass das zu entwickelnde Softwaresystem mehrere Nutzer potentiell gleichzeitig bei den Geschäftsprozessen der Vormerkung, Ausleihe und Rückgabe von Medien einer Bibliothek unterstützen soll. Zu diesem Zweck verwaltet das System die Leserdaten und den Medienbestand einer Bibliothek.

Leser und Bibliotheksangestellte stellen die beiden Arten von Nutzern des Systems dar. Für Leser sei lediglich die Durchführung von Vormerkungen zulässig, Bibliotheksangestellte sollen jedoch die Möglichkeit haben alle Geschäftsprozesse durchzuführen.

Das Bibliothekssystem soll hinsichtlich der unterstützten Geschäftsprozesse einfach erweiter- und änderbar sein.

2 Systemmodell

2.1 Einleitung

Unter einem Entwicklungsprozess verstehen wir die schrittweise Erarbeitung von Entwicklungsinhalten. Beispielsweise dienen die ersten Entwicklungsschritte eines Forward-Engineering-Vorgehens der Erarbeitung der zu lösenden fachlichen Aufgabenstellung. Die daran anschließenden Entwicklungsschritte haben die Erarbeitung einer softwaregestützten Lösung dieser Aufgabenstellung zum Ziel.

Voraussetzung für einen systematischen und bruchfreien Entwicklungsprozess sind aufeinander abgestimmte Entwicklungsschritte. Das setzt wiederum voraus, dass die zu erbringenden Ergebnisse, das heißt die Entwicklungsprodukte, aufeinander abgestimmt und die Zusammenhänge zwischen ihnen eindeutig festgelegt sind. Beispielsweise müssen Anforderungsspezifikation und Software-Architektur so aufeinander abgestimmt sein, dass klare Kriterien existieren, die angeben, wann eine Software-Architektur eine Anforderungsspezifikation erfüllt.

Im wesentlichen existieren bereits zwei unterschiedliche Ansätze, um inhaltliche Zusammenhänge zwischen Entwicklungsprodukten anzugeben:

- Ein Ansatz ist die Angabe von Zusammenhängen jeweils für Paare einer gegebenen Menge von Produktarten, beispielsweise zwischen Spezifikationen von Ablaufsequenzen (Systemlebenszyklen) und Automatenspezifikationen eines Systems. Der sogenannte *viewpoint*-Ansatz [NKF94] ist von dieser Art.
- Eine andere Möglichkeit ist die, *ein* umfassendes Modell *aller* Entwicklungsinhalte festzulegen, und die Bedeutung aller Produktarten anhand dieses einen Modells anzugeben. Produktzusammenhänge ergeben sich damit aus den im Modell festgelegten Zusammenhängen zwischen den jeweiligen Produktinhalten. [Bre98, Huß97, Rum96] sind Beispiele für Arbeiten, in denen dieser Ansatz verfolgt wird.

Wir wählen letztere Variante, da sie die weitreichendere darstellt: Paarweise Produktbeziehungen können aus den Beziehungen ihrer Inhalte schlüssig abgeleitet werden. Ein sinnvolles Modell vorausgesetzt, dienen die im Modell festgelegten inhaltlichen Zusammenhänge quasi der Begründung von Produktbeziehungen.

Unser Modell der Entwicklungsinhalte nennen wir das *Systemmodell* (der Entwicklung). Dieses Modell dient zum einen, wie erwähnt, dazu, Entwicklungsprodukten eine Bedeutung zu geben. Da wir das *eine* Modell für *alle* Produkte verwenden, ergeben sich Zusammenhänge zwischen Produkten aus den im Systemmodell festgelegten Zusammenhängen zwischen den Inhalten, welche die einzelnen Produkte beschreiben (siehe Abbildung 2.1). Zum zweiten stellt das Systemmodell allen an der Entwicklung Beteiligten eine *gemeinsame* Begriffswelt/Ontologie zur Verfügung. Ähnlich zu Entwurfsmustern [GHJ+95] hilft diese gemeinsame Begriffswelt Missverständnisse zu vermeiden. Auch eine präzise Definition des Systemmodells dient diesem Ziel. Wir erreichen Präzision durch die mathematische Formulierung des Systemmodells.

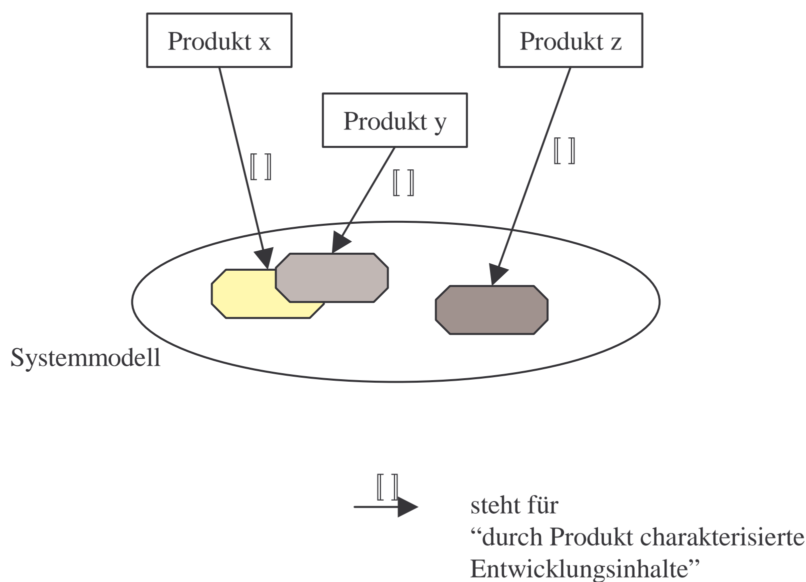


Abbildung 2.1: Bedeutung von Produkten anhand eines gemeinsamen Systemmodells.

Wir halten fest:

Das Systemmodell ist ein mathematisches Modell der Inhalte einer Softwareentwicklung.

Beispiel 2.1 (Festlegung von Produktzusammenhängen anhand eines gemeinsamen Systemmodells ihrer Inhalte)

Das Produkt der „Spezifikation des fachlichen Zustandsraums“ (häufig auch als fachliche oder konzeptuelle Datenspezifikation bezeichnet) lege den durch ein Softwaresystem zu realisierenden fachlichen Zustandsraum fest. Das Produkt der „Software-Architektur“ lege die Eigenschaften des Softwaresystems, modelliert als Komponentenetzwerk, fest.

Sinnvollerweise muss das Komponentennetzwerk so gestaltet sein, dass es den fachlichen Zustandsraum „realisiert“. Im Systemmodell legen wir daher diese „Re-

alisierungsbeziehung“ präzise fest, indem wir die Existenz einer Ähnlichkeitsabbildung zwischen fachlichem Zustandsraum und dem durch die Komponenten des Netzwerkes aufgespannten Zustandsraums fordern

Eine fachliche Zustandsraumspezifikation und eine Software-Architektur betrachten wir dann als widerspruchsfrei/konform, falls ein System existiert, das sowohl die in den beiden Produkten formulierten Eigenschaften aufweist, als auch die im Systemmodell gegebenen Realisierungsbedingungen erfüllt.

□

Die im Systemmodell zu definierenden Entwicklungsinhalte ergeben sich aus den drei Entwicklungsaufgaben, die wir, gemäss der in Abschnitt 1.2 gegebenen Zielsetzung, in dieser Arbeit behandeln:

- **Fachliche Aufgabenstellung.** Unternehmen, oder Organisationen im allgemeinen, bilden den typischen Anwendungskontext von Informationssystemen. Die typischen Aufgaben von Informationssystemen sind die Verwaltung fachlicher Objekte/fachlicher Information und die Unterstützung fachlicher Abläufe. Dementsprechend enthält das Systemmodell Elemente zur Charakterisierung fachlicher Objekte und Abläufe.
- **Softwaresystem-Entwurf.** Die Aufgabe des Softwaresystem-Entwurfs besteht in der Festlegung von strukturellen und Verhaltenseigenschaften des zu entwickelnden Softwaresystems. Wir konzentrieren uns in dieser Arbeit auf verteilte Informationssysteme, so dass ein in autonom agierende und interagierende Komponenten strukturiertes Modell verteilter Systeme, Teil des Systemmodells ist. Anhand dessen charakterisieren wir Innen- und Außensicht, sowohl des Softwaresystems als auch von dessen Umgebung.
- **Anforderungsspezifikation.** Wir sehen die Aufteilung einer fachlichen Aufgabenstellung auf das (zu entwickelnde) Softwaresystem und dessen Umgebung als zentralen Bestandteil der Anforderungsspezifikation. Dadurch legen wir fest, welche fachlichen Objekte und (Anteile von) fachlichen Abläufen durch das Softwaresystem beziehungsweise durch dessen Umgebung zu realisieren sind. Das Systemmodell enthält Elemente zur Charakterisierung dieser Realisierungsbeziehung. Mit den im Systemmodell festgelegten, immer einzuhaltenden, grundlegenden Eigenschaften dieser Realisierungsbeziehung, definieren wir Kriterien, die angeben, wann ein aus Softwaresystem und Umgebung aufgebautes Komponentennetzwerk als Realisierung einer durch Objekte und Abläufe gegebenen Aufgabenstellung angesehen werden kann.

Neben der Frage nach den Inhalten, stellt sich auch die Frage, was es bedeutet, ein *Modell* dieser Inhalte anzugeben ?

Allgemein steht der Begriff des Modells für *Vorbild, Muster und Entwurf*. Unser Systemmodell ist Entwurf, Vorbild und Muster für die Inhalte einer spezifischen Entwicklung, da es eine Menge von (mathematischen) Elementen und Eigenschaften dieser Elemente vorgibt, die wir so gewählt haben, dass die Inhalte einer spezifischen Entwicklung unter Verwendung und durch Konkretisierung dieser Elemente erfasst werden können. Die im Systemmodell vorgegebenen Eigenschaften dienen dazu, nicht sinnvolle/inkonsistente Entwicklungsinhalte auszuschließen, so

dass die Inhalte einer spezifischen Entwicklung diesen Eigenschaften genügen müssen (vergleiche Beispiel 2.1). Um das Modell präzise angeben zu können, verwenden wir mathematische Konzepte. Das Modell wählen wir so, dass es angemessen für die Softwareentwicklung ist; das bedeutet beispielsweise eine Sicht auf die Anwendungswelt, die möglicherweise für eine andere Aufgabe als die der Softwareentwicklung, zum Beispiel für die umfassende Gestaltung von Arbeitsabläufen in einem Unternehmen, nur eingeschränkt geeignet oder zumindest entsprechend zu erweitert ist.

Warum nennen wir unser Modell ein *System*modell ?

Der Begriff des Systems steht im allgemeinen für ein *sinnvoll in sich gegliedertes, geordnetes Ganzes*. Da in unserem (System-)Modell die Entwicklungsinhalte nicht zusammenhangslos nebeneinander gestellt werden, sondern immer in einem klaren Bezug zueinander stehen um ein sinnvolles Ganzes zu ergeben, sprechen wir von einem Systemmodell (der Entwicklungsinhalte).

Die folgende Einführung des Systemmodells gliedert sich wie folgt:

Zunächst gehen wir darauf ein, was wir im Detail darunter verstehen, Entwicklungsprodukten eine Bedeutung anhand eines Systemmodells zu geben. Das Prinzip der losen Interpretationen/Unterspezifikation ist hier zentral.

Im Anschluss daran geben wir die Systemmodellelemente und ihre Eigenschaften an, wobei wir das Systemmodell gliedern in

- die *ganzheitliche Perspektive*, welche die Elemente zur Beschreibung der fachlichen Aufgabenstellung einer Entwicklung zusammenfasst. Diese Perspektive nennen wir ganzheitlich, da hier die Aufgabenstellung im Vordergrund steht, und nicht die Verteilung von Aufgaben auf Aufgabenträger, insbesondere auf den Aufgabenträger, den das zu entwickelnde Softwaresystem darstellt. Die zentralen Modellierungskonzepte sind in dieser Perspektive, wie bereits erwähnt, fachliche Objekte, die wir im folgenden auch Entitäten nennen, sowie fachliche Abläufe.
- die *komponentenbasierte Perspektive*, zur Beschreibung des Softwaresystems einer Entwicklung und dessen Umgebung. Softwaresystem und Umgebung bilden ein aus Komponenten aufgebautes Netzwerk, wobei wir unter Komponenten autonom agierende, durch Nachrichtenaustausch interagierende Einheiten verstehen. Durch Komponenten modellieren wir die Aufgabenträger des betrachteten Anwendungssystems.
- die *Perspektiven-Zusammenhänge*. Damit fassen wir die Systemmodell-Elemente zusammen, mittels derer wir beschreiben, auf welche Weise das Softwaresystem und seine Umgebung die fachliche Aufgabenstellung realisieren, etwa durch welche Interaktionsmuster fachliche Abläufe realisiert werden.

Der Unterschied zwischen ganzheitlicher und komponentenbasierter Perspektive, lässt sich zusammenfassend wie folgt formulieren:

Während in der ganzheitlichen Perspektive angegeben wird, welche Entitäten und welche fachlichen Abläufe zu realisieren sind, werden in der komponentenbasier-

ten Perspektive die Elemente (in unserem Fall die Komponenten) angegeben, die Entitäten abbilden und Abläufe „generieren“.

Sowohl in der ganzheitlichen, als auch in der komponentenbasierten Perspektive beziehen wir uns auf einen für die Softwareentwicklung relevanten Ausschnitt eines Unternehmens oder einer Organisation im allgemeinen. Diesen Unternehmensausschnitt bezeichnen wir im Folgenden als das *Anwendungssystem* (eines zu entwickelnden Softwaresystems).

2.2 Systemmodell, Systeme und Entwicklungsprodukte

Vorgehensmodelle machen Vorgaben für die Durchführung einer Softwareentwicklung. Diese Vorgaben sollen dabei helfen, (zusammen)passende Entwicklungsinhalte auf systematische Weise zu erarbeiten und für die Formulierung der Inhalte geeignete Konzepte zu verwenden, um so effizient qualitativ hochwertige Software zu entwickeln.

Wie oben beschrieben, gibt unser Systemmodell Entwicklungsinhalte vor, die wir zur Entwicklung verteilter Informationssysteme für geeignet halten. Bezeichnen wir die im Rahmen einer Entwicklung erarbeiteten Inhalte als *Entwicklungssystem*, dann können wir das Systemmodell als die Menge aller Entwicklungssysteme verstehen, die wir als sinnvoll für die Entwicklung verteilter Informationssysteme erachten. Beispielsweise ist ein Entwicklungssystem nur dann „sinnvoll“, wenn das im Entwicklungssystem beschriebene Softwaresystem auch die im Entwicklungssystem festgelegten fachlichen Anforderungen erfüllt.

Zweck eines *Entwicklungsproduktes* ist es, bestimmte Inhalte, das heißt bestimmte Eigenschaften eines Entwicklungssystems, festzulegen. Zum Beispiel legen wir mit einer Anforderungsspezifikation die fachlichen Eigenschaften fest, welche das Softwaresystem zu erfüllen hat. Andere Inhalte, wie zum Beispiel der interne Aufbau des Softwaresystems, werden durch eine Anforderungsspezifikation nicht beschrieben und damit offen gelassen.

Das Prinzip, bestimmte Inhalte festzulegen und andere offen zu lassen, bestimmt unsere Interpretation von Entwicklungsprodukten: Wir wählen die Bedeutung von Entwicklungsprodukten als die Menge aller Entwicklungssysteme, welche die im Produkt festgelegten Eigenschaften erfüllen. All die Inhalte von Entwicklungssystemen, die nicht durch ein Produkt festgelegt werden, können, im durch das Systemmodell vorgegebenen Rahmen, beliebige Eigenschaften besitzen. Damit interpretieren wir Produkte im Sinne der Unterspezifikation: nicht explizit festgelegte Inhalte können beliebig ausgeprägt sein. Wir kennen diesen Ansatz beispielsweise auch von der losen Semantik algebraischer Spezifikationen (siehe etwa [Wir90]). Mit der losen Semantik von Produkten ergibt sich die Bedeutung einer Menge von Produkten auf einfache Weise durch Schnittmengenbildung.

Wir halten fest:

Das Systemmodell ist die Menge aller „sinnvollen“ Entwicklungssysteme, das heißt der Systeme von Entwicklungsinhalten.

lose Semantik:

Die Bedeutung/Semantik eines Entwicklungsproduktes ist gleich der Menge aller Entwicklungssysteme des Systemmodells, welche die im Produkt formulierten Eigenschaften erfüllen.

Formal fassen wir die Begriffe *Systemmodell* und *(Entwicklungs-)System* sowie die Semantik von Entwicklungsprodukten in Definition 2.1, Definition 2.2 und Definition 2.4. Diese Definitionen bauen auf der Beschreibung der Systemmodell-inhalte durch mathematische Elemente auf. Zum Beispiel modellieren wir den fachlichen Zustandsraum eines Entwicklungssystems durch eine Rechenstruktur, die eine Sorte der Zustände umfasst. Abbildung 2.2 fasst die Systemmodellelemente zusammen.

V	Menge der fachlichen Wertsorten
Σ_{fe}	fachliche Entitätssignatur
$C_f \subseteq Gen(\Sigma_{fe})$	Klasse der fachlichen Rechenstrukturen[2]
$A \in C_f$	fachliche Rechenstruktur
$E \subseteq A_{ID}$	Menge der Entitäten
Σ_{fs}	fachliche Zustandssignatur
$FS \in Alg(\Sigma_{fs})$	fachliche Zustandsalgebra
$init_state \in FS_{State}$	initialer Systemzustand[3]
M	Menge der Ereignismarkierungen
$P \subseteq [pcs_{fin}(EV,M)]$	Menge der Systemprozesse[4]
$reak : M \to \wp([pcs_{fin}(EV,M)])$	Reaktionsabbildung
N	Nachrichtenuniversum
$ctype : CH \to \wp(N)$	Kanaltypisierung
$nw, dnw, enw \in NW(CH)$[5], $nw = dnw \cup enw$	Realisierungsnetzwerk, bestehend aus Entwicklungs- und Umgebungsnetzwerk
$n_{snw} \in \mathbb{N}_+$	Anzahl der Spezifikationsnetzwerke
$K_i \subseteq KID, i \in [1, ..., n_{snw}]$	Bezeichner der Komponenten einzelner Spezifikationsnetzwerke
$K \subseteq KID$, $K = \bigcup_{i \in [1,\ldots,n_{snw}]} K_i$	Bezeichner der Komponenten aller Spezifikationsnetzwerke

[2] $Gen(\Sigma)$ steht für die Menge der Σ-Rechenstrukturen (siehe Definition 8.6).

[3] für eine Sorte s einer Algebra X bezeichnen wir mit X_s die Trägermenge der Sorte s in X (vergleiche Definition 8.3)

[4] $pcs_{fin}(EV,M)$ steht für die Menge der endlich fundierten Prozesse über dem Ereignisuniversum EV und der Markierungsmenge M; mit $[pcs_{fin}(EV,M)]$ bezeichnen wir die Isomorphieklassen über $pcs_{fin}(EV,M)$ (vergleiche Definition 8.9 bis Definition 8.13).

[5] mit NW(CH) bezeichnen wir die Menge der Netzwerke über (der Kanalmenge) CH (siehe Definition 8.21).

$comp : K \rightarrow TKP(CH)$	Topologiekomponenten[6] der Spezifikationsnetzwerke
$develop : K \rightarrow \mathbb{B}$	Entwicklungsklassifikation
$snw_i, dnw_i, enw_i \in TNW(CH)$, $snw_i = dnw_i \cup enw_i$	Spezifikationsnetzwerke, jeweils bestehend aus Entwicklungs- und Umgebungsnetzwerk[7]
I_k , O_k	Ein- und Ausgabekanäle,
$behavior_k : \overrightarrow{I_k} \rightarrow \wp\left(\overrightarrow{O_k}\right)$ [8]	Verhaltensfunktion,
$tclass_k \in TClass$[9]	Topologieklassifikation,
$states_k$	Zustände,
$exec_k \subseteq \left(IS_{I_k \cup O_k, states_k}\right)^{\infty}$	Ausführungsfolgen[10],
STM_k	Automat,
$inits_k \subseteq states_k \times \overrightarrow{O_k}^{*}$ [11]	Initialelemente und
$\delta_k : \left(states_k \times \overrightarrow{I_k}^{*}\right) \rightarrow \wp\left(states_k \times \overrightarrow{O_k}^{*}\right)$	Transitionsfunktion des Automaten,
	der mit $k \in K$ bezeichneten (Topologie-) Komponente.
Σ_i	Zustandssignatur,
$C_i \subseteq Alg(\Sigma_i)$	Klasse der Zustandsalgebren,

[6] TKP(CH) steht für die Menge aller Topologiekomponenten über der Kanalmenge CH (siehe Definition 2.17).

[7] TNW(CH) steht für die Menge aller Topologienetzwerke über der Kanalmenge CH (siehe Definition 2.18).

[8] für eine Kanalmenge X steht $\overrightarrow{X}$ für die Kanalbelegungen über X (vergleiche Definition 8.18)

[9] *TClass* steht für die Menge *{behavior, interface, real}* der topologischen Klassen (vergleiche Definition 2.17)

[10] für eine Kanalmenge C und eine Zustandsmenge S steht $IS_{C,S}$ (Interaktions-Zustands-Tupel) für $\overrightarrow{C}^{*} \times S$, wobei $\overrightarrow{C}^{*}$ für die Menge der Interaktionsmuster über C steht (vergleiche Definition 8.22 und Definition 8.18).

[11] für eine Kanalmenge X steht $\overrightarrow{X}^{*}$ für die Interaktionsmuster über X.

$KS_i \in C_i$	Zustandsalgebra,
$IS_i = IS_{channels(snw_i),states(snw_i)}$	Interaktions-Zustands-Tupel,
$isreak_i : (IS_i)^* \rightarrow \wp((IS_i)^\omega)$	Reaktionsabbildung,
	für jedes Spezifikationsnetzwerkes snw_i, $i \in [1,...,n_{snw}]$
$\sigma_i : \Sigma_{fs} \rightarrow \Sigma_i$	Signaturmorphismus,
$hs_i : (KS_i)\|_{\sigma_i} \rightarrow FS$	Homomorphismus der Zustandsalgebren,
$hi_i : M \rightarrow \wp((IS_i)^\omega)$	Interaktionszuordnung,
$GStreams_i \subseteq (IS_i)^\omega$	zulässige (aus P mittels hi_i abgeleitete) Ausführungsfolgen,
	für jedes Spezifikationsnetzwerk snw_i,
	$i \in [1,...,n_{snw}]$

Abbildung 2.2 (Begriffe des Systemmodells)

Definition 2.1 (Systemmodell)

Als *Systemmodell* bezeichnen wir die Gesamtheit, der in Abbildung 2.2 aufgeführten, und in Definition 2.6 bis Definition 2.30 festgelegten Begriffe und ihrer Zusammenhänge.

Als gegeben nehmen wir die drei folgenden, nichtleeren Mengen an, die grundlegend für alle Systeme sind:

EV	Universum der Ereignisbezeichner
CH	Kanaluniversum
KID	Universum der Komponentenbezeichner

□

Definition 2.2 (System)

Ein *System* ist eine Ausprägung der Begriffe des Systemmodells, so dass alle (in Definition 2.6 bis Definition 2.30) definierten Eigenschaften und Zusammenhänge erfüllt sind.

Die *Menge der Systeme* bezeichnen wir mit *SYS*.

Formal ist damit ein System sys $\in$ SYS ein Tupel, bestehend aus den in Abbildung 2.2 aufgeführten Elementen. Zur Selektion eines dieser Elemente verwenden wir die *Subskriptionsschreibweise*. So bezeichnen wir beispielsweise mit $(C_f)_{sys}$ die Klasse der Entitätsrechenstrukturen (C_f) eines Systems *sys*.

□

Entwicklungsprodukte verstehen wir als Aussagen über Systeme, so dass wir die Menge der Systeme, die diese Aussage erfüllen, als die Bedeutung eines Produktes verstehen. Zur sinnvollen Strukturierung von Entwicklungsprodukten definieren wir in dieser Arbeit eine Reihe von *Entwicklungsproduktarten*, wobei wir eine Produktart als ein Spezifikationsschema verstehen. Jedes dieser Schemata charakterisieren wir durch einen eindeutigen Bezeichner (der Produktart) sowie durch die Elemente, aus denen eine dem Schema entsprechende Spezifikation aufgebaut ist. Diese Elemente definieren sozusagen die abstrakte Syntax einer Produktart. Notationen, die geeignete Repräsentationen dieser Elemente anbieten, liefern die konkrete Syntax von Entwicklungsprodukten. Mathematisch entspricht die Festlegung der Elemente einer Produktart einem n-fachen kartesischen Produkt.

Definition 2.3 (Entwicklungsproduktarten und Menge der Entwicklungsprodukte)

In Definition 2.5 sowie in den Definitionen von Kapitel 3 bis Kapitel 6 legen wir die Produktarten dieser Arbeit fest. Die Menge *EP* der (in dieser Arbeit betrachteten) Entwicklungsprodukte, verstehen wir als die Menge aller Paare, die aus einem Produktart-Bezeichner und einem, der Produktart entsprechenden n-Tupel von Produktartelementen bestehen.

Ein Produkt einer Entwicklungsproduktart *x*, deren abstrakte Syntax durch ein n-faches kartesisches Produkt gegeben ist, beschreiben wir durch

$(a1, \ldots, an)_x$.

□

Definition 2.4 (Semantik von Entwicklungsprodukten)

Die Semantik von Produkten beschreiben wir durch die Abbildung

$[\![.]\!] : \mathrm{EP} \rightarrow \wp(\mathrm{SYS})$.

Die Semantik einer Menge $P \subseteq EP$ von Produkten entspricht aus logischer Sicht der Konjunktion der Aussagen der einzelnen Produkte aus P, was mengentheoretisch der Schnittmengenbildung entspricht:

$[\![.]\!]^* : \wp(\mathrm{EP}) \rightarrow \wp(\mathrm{SYS})$

mit

$$[\![P]\!]^* =_{\mathrm{def}} \bigcap_{p \in P} [\![p]\!]$$

Wir schreiben im Folgenden auch $[\![P]\!]$ für $[\![P]\!]^*$.

□

Einzelne Produkte können wir als partielle Spezifikationen verstehen. Die in [ZJ93] zur Komposition von (partiellen) Spezifikationen formulierten Gedanken finden wir, in übertragener Form, mit den obigen Definitionen in unserem Ansatz wieder.

Die allgemeinste Entwicklungsproduktart können wir bereits an dieser Stelle, vor der Definition des Systemmodells angeben, da wir, im Unterschied zu den in Kapitel 3 bis Kapitel 6 definierten Produktarten, nicht auf bestimmte Systemmodellelemente Bezug nehmen. Diese allgemeinste Art von Entwicklungsprodukt besteht aus einem (beliebigen) Prädikat über der Menge SYS. Aufgrund seiner Allgemeinheit bietet diese Produktart keine methodische Hilfestellung. Wir benötigen diese, als *Entwicklungsprädikat* bezeichnete Produktart dennoch, um spezifischere Entwicklungsprodukte in einen gemeinsamen Kontext einer Spezifikation einbetten zu können, und um Bezüge zwischen Elementen unterschiedlicher Entwicklungsprodukte herstellen zu können, die Bausteine einer aus mehreren Entwicklungsprodukten aufgebauten Spezifikation darstellen (vergleiche etwa Beispiel 4.1, wo wir Verhaltensspezifikationen in Bezug zu Datenspezifikationen setzen).

Definition 2.5 (Entwicklungsprädikat)

Ein *Entwicklungsprädikat* ist ein 1-Tupel, bestehend aus

- einem Prädikat $p : SYS \rightarrow \mathbb{B}$.

Die Semantik definieren wir wie folgt:

$$[\![(p)_{pred}]\!] =_{def} \{ sys \in SYS | p(sys) \}$$

□

Um einen Überblick zu geben, skizzieren wir grob die Inhalte des Systemmodells, die wir in den folgenden Abschnitten einführen und die sich, wie oben erwähnt, in drei Teile gliedern lassen:

- **Ganzheitliche Perspektive.**

  Die ganzheitliche Perspektive (des Anwendungssystems) umfasst die Systemmodellanteile, die zur Beschreibung der fachlichen Aufgabenstellung dienen, die mit Hilfe des zu entwickelnden Softwaresystems zu lösen ist. In dieser Perspektive charakterisieren wir das Anwendungssystem durch einen globalen Zustandsraum und eine Menge von Prozessen. Die Prozessmenge steht für die möglichen Lebenszyklen/Abläufe des Anwendungssystems (Nichtdeterminismus). Die Prozesse eines Bibliothekssystems, beispielsweise, setzen sich aus Vormerkungs-, Entleihe- und Rückgabevorgängen zusammen.

  Den Zustandsraum des Anwendungssystems spannen wir in der ganzheitlichen Perspektive durch eine Menge von Entitäten auf. Durch Entitäten modellieren wir die Objekte der Anwendungswelt. Wir modellieren Entitäten als Zustandsvariablen. Durch Klassifikation der Entitäten legen wir fest, welche (Zustands-)werte Entitäten einnehmen können. Beispielsweise bilden Medien- und Leserentitäten den Zustandsraum einer Bibliothek.

  Unter einem Prozess verstehen wir eine Menge von Ereignissen und ihre kausalen Abhängigkeiten. Durch die Kausalität von Ereignissen ergibt sich eine zeitliche Reihenfolge. Beispielsweise ist die Äußerung eines Vormerkungswunsches kausal für die Zuteilung eines Mediums an einen Leser.

  Um Ereignissen eine nähere Bedeutung geben zu können, unterscheiden wir drei Arten von Ereignissen:

  - Zustandsänderungen
  - Zustandseigenschaften/Zustandsprädikate
  - Aktionen, die in der ganzheitlichen Perspektive nicht weiter interpretiert werden, aber in der komponentenbasierten Perspektive als Bezeichner für bestimmte Verhaltensmuster verstanden werden.

  Durch Zustandsänderungs- und Zustandseigenschaftsereignisse ergibt sich der Zusammenhang zwischen Zustand und Verhalten. Die Rückgabe eines Mediums erfassen wir beispielsweise durch eine Zustandsänderung, da sich mit der Rückgabe der Zustand des jeweiligen Mediums verändert (der Status des Mediums wechselt von *entliehen* auf *verfügbar*).

Ereignisse, die mit Zustandsprädikaten markiert sind, entsprechen der Abfrage von Eigenschaften des Systemzustands. Der Verlauf von Geschäftsprozessen kann von dem Ergebnis solcher Abfragen abhängen. Zum Beispiel wird in einem Entleih-Prozess das gewünschte Medium nur dann ausgegeben, wenn es verfügbar ist.

Abbildung 2.3 illustriert die Elemente der ganzheitlichen Perspektive anhand eines exemplarischen Prozesses, der Vormerkung eines Mediums in einer Bibliothek. Den Prozess stellen wir als Aktivitätsdiagramm der UML [OMG01] dar. Die in den Prozessereignissen referenzierten Entitäten *m* und l (einer Medien beziehungsweise Leser-Entität) erfassen wir durch ein Objektdiagramm der UML [OMG01]. Wir gehen davon aus, dass für Medien-Entitäten die Operationen *status* und *vormerken* definiert sind, und *verfügbar* sowie *entliehen* mögliche (Rückgabe-)Werte der Operation *status* sind. Einen Entitätsbezug von Ereignis *vormerkungswunsch(m,l)* ist in der Abbildung grau hinterlegt illustriert. Die Markierung *vormerkungswunsch(m,l)* ist ein Beispiel für eine Aktionsmarkierung, *m.vormerken*(l)* für eine Zustandsänderung und *m.status()==verfügbar* für eine Zustandseigenschaft.

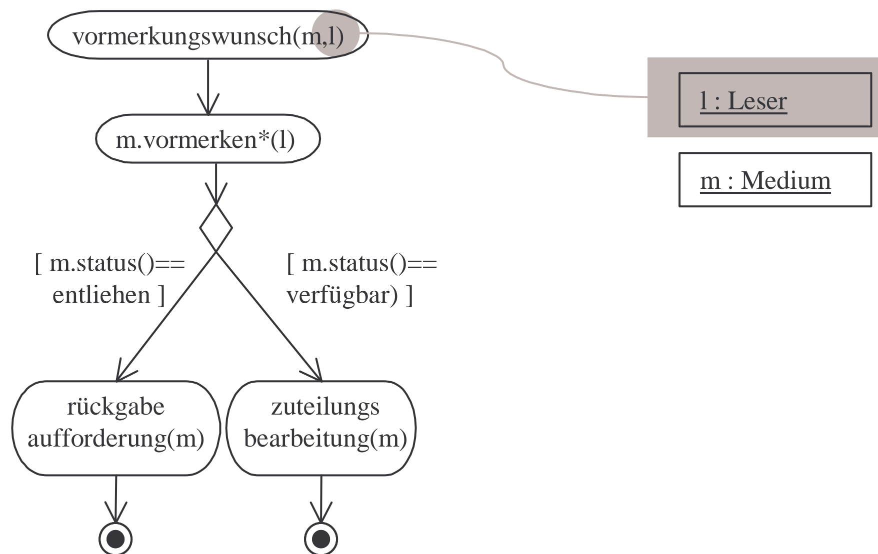


Abbildung 2.3: Skizze der ganzheitlichen Perspektive anhand eines exemplarischen Aktivitäts- und Objektdiagramms.

- **Komponentenbasierte Perspektive.**

  Die komponentenbasierte Perspektive des Anwendungssystems dient der Beschreibung des zu entwickelnden Softwaresystems und dessen Umgebung. Im Unterschied zur ganzheitlichen Perspektive, verteilen wir, in der komponentenbasierten Perspektive, Zustands- und Verhaltensanteile auf eine endliche Menge von Komponenten.

  Jede Komponente kapselt einen Zustandsraum. Änderungen des Zustands einer Komponente sowie Kontrollfluss zwischen Komponenten setzt explizite Kommunikation zwischen Komponenten voraus. Diese explizite

Kommunikation modellieren wir, indem wir Komponenten über gerichtete Kanäle zu einem Netzwerk verbinden. Über diese Kanäle tauschen Komponenten Nachrichten aus. Beispielsweise können wir ein Bibliothekssystem aus einer Kontroll- und einer Datenbankkomponente aufbauen (siehe Abbildung 2.4, in der die beiden Komponenten *Kontrolle* und *Datenbank* durch die Kanäle *dbk* und *kdb* zu einem Netzwerk verbunden sind). Die Kontrollkomponente hat die Aufgabe, Abläufe zu steuern. Im Rahmen der Bearbeitung eines Vormerkungswunsches, sendet die Kontrollkomponente eine Nachricht an die Datenbankkomponente, um die Vormerkung des Mediums einzutragen.

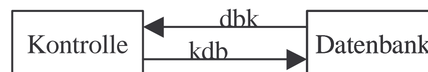


Abbildung 2.4: Netzwerk, bestehend aus zwei Komponenten, die über zwei Kanäle verbunden sind.

- **Perspektiven-Zusammenhänge.**

  Unter diesem Begriff fassen wir die Systemmodellelemente zusammen, durch die wir angeben, auf welche Weise das durch Softwaresystem und dessen Umgebung gebildete Komponentennetzwerk den globalen Zustandsraum und das Prozessverhalten der ganzheitlichen Perspektive realisiert.

  Dazu sind zum einen Operationen auf dem durch das Netzwerk aufgespannten Zustandsraum anzugeben, welche den Operationen entsprechen, die auf dem globalen Zustandsraum der ganzheitlichen Perspektive definiert sind. Da wir in beiden Perspektiven den Zustandsraum und die darauf definierten Operationen in Form von Rechenstrukturen modellieren, beschreiben wir die Zustandsraumrealisierung in Form eines Rechenstruktur-Homomorphismus (vergleiche Definition 8.5).

  Um die Realisierung von Prozessen erfassen zu können, müssen wir Prozessereignisse und Komponenteninteraktionen zueinander in Bezug setzen. Dazu geben wir eine Abbildung an, die zu jeder Art von Prozessereignis die Interaktionsmuster angibt, die eine Realisierung dieser Ereignisart darstellen.

  Beispielsweise können wir die Medienentitäten einer Bibliothek in einer Mediendatenbankkomponente realisieren. Das Ereignis, welches die Äußerung eines Vormerkungswunsches darstellt, bilden wir dann ab, durch das Senden einer Nachricht von der Umgebung an die Kontrollkomponente, die den Vormerkungswunsch erfasst, gefolgt von einer entsprechenden Nachricht von der Kontroll- an die Datenbankkomponente, zur Eintragung der Vormerkung. Das Sequenzdiagramm in Abbildung 2.5 illustriert das Beispiel.

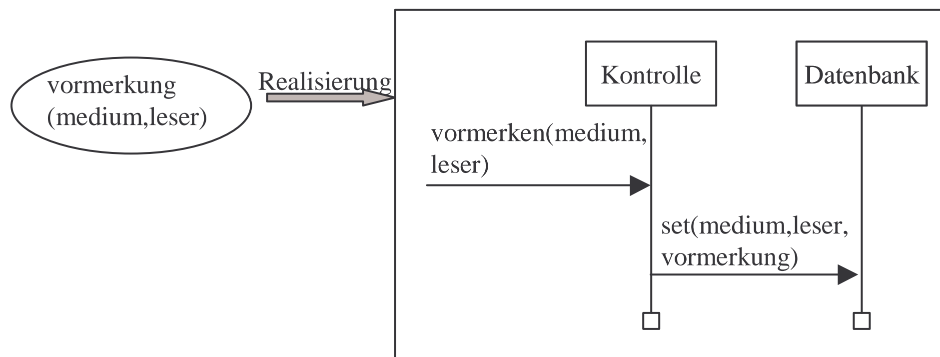


Abbildung 2.5: Vormerkungsereignis und seine Realisierung

2.3 Ganzheitliche Perspektive

Als ganzheitliche Perspektive bezeichnen wir den Teil des Systemmodells, der zur Beschreibung der fachlichen Aufgabenstellung dient, die mit Hilfe des zu entwickelnden Softwaresystems zu realisieren ist. Wie in Abschnitt 1.2 motiviert, ist die Beschreibung der fachlichen Aufgabenstellung ein wichtiger Bestandteil einer Softwareentwicklung. Sie dient einem klaren Verständnis der fachlichen Rolle des Softwaresystems.

Der typische Anwendungskontext von Informationssystemen sind Unternehmen. Typische Aufgabenstellungen für Informationssysteme sind das Verwalten fachlicher Information und die Unterstützung fachlicher Abläufe. Beispielsweise dient ein Informationssystem der Verwaltung von Konten und der Unterstützung von Überweisungsvorgängen.

Die Analyse von Ansätzen zur Beschreibung von Unternehmen im Zusammenhang mit einer Softwareentwicklung [Dav93, Kos62, Pae98b, Schee98, WFMC99] zeigt, dass zentrale Betrachtungsgegenstände die

- Objekte der Anwendungswelt, die bearbeitet werden, zum Beispiel Konten, Dokumente und Werkstücke, sowie
- die Abläufe der Anwendungswelt

sind. Objekte der Anwendungswelt bezeichnen wir im Folgenden auch als fachliche Entitäten, in Anlehnung an das Konzept der Entitäten, wie es aus der Entity-Relationship-Modellierung [Che76], die etwa in [Schee98] und SSADM[DCC92] zur Spezifikation von Objekten der Anwendungswelt eingesetzt wird, bekannt ist. Neben der Beschreibung von Entitäten und Abläufen ist auch die Beschreibung der Aufgabenverteilung, das heißt die Zuordnung von Entitäten und (Anteilen von) Abläufen zu Aufgabenträgern/Rollen/Organisationseinheiten, notwendig, jedoch sekundär. Folgendes Zitat aus [Schee96] verdeutlicht dies:

> *„Trotz einer Vielzahl unterschiedlicher Reorganisationsprojekte haben sich in den letzten Jahren die Geschäftsprozesse als zentraler Betrachtungsgegenstand organisatorischer Umgestaltungen herausgebildet. [...]*

> *Neben der Beschreibung der Ablaufstruktur aus Ereignissen und Funktionen ist auch die Angabe der den Funktionen zugeordneten Organisationseinheiten von Interesse. Viele Reorganisationsprojekte beziehen sich gerade auf eine neue Zuordnung von Funktionen zu Organisationseinheiten.“*

Obiges Zitat und die Thematik der Unternehmens-Reorganisation (Business Reengineering) im allgemeinen machen deutlich, dass Abläufe, die im Kern gleich sind, mit unterschiedlichen Aufgabenverteilungen realisiert werden können. Das bedeutet, dass die Aufgabenverteilung nicht als Teil der fachlichen Aufgabenstellung, sondern als Teil der Lösung zu sehen ist. Insbesondere ist mit der fachlichen Aufgabenstellung im allgemeinen noch nicht die Grenze zwischen zu entwickelndem Softwaresystem, das einen speziellen Aufgabenträger darstellt, und dessen Anwendungsumgebung gegeben. Beispielsweise kann die fachliche Aufgabe der Versendung einer E-Mail (zur Benachrichtigung eines Lesers bei Verfügbarkeit eines vorgemerkten Mediums) sowohl dem Softwaresystem als auch dem Bibliothekspersonal zur Aufgabe gemacht werden.

Die zentrale Bedeutung einer umfassenden, von Aufgabenträgern unabhängigen, Erfassung fachlicher Abläufe ist auch in [BDI97] zu erkennen, wo sie als Grundlage für die Festlegung der Funktionalität des zu entwickelnden (Informations-)Systems dient:

> *„Zur Festlegung der geforderten Systemfunktionalität werden Geschäftsprozesse definiert. Diese sind ganzheitlich, d. h. vollständig zu betrachten, auch wenn das geforderte System nur Ausschnitte davon unterstützt. [...] Das System wird zusammen mit seiner Umgebung (Mensch, verbundene Systeme) als Ganzes gesehen.“*
>
> (vergleiche Beschreibung des *Anwenderforderungen* genannten Entwicklungsprodukts in [BDI97]).

Aus obigen Betrachtungen ergibt sich die Notwendigkeit, die ganzheitliche Perspektive unseres Systemmodells so zu wählen, dass wir Entitäten und Abläufe des Anwendungssystems *unabhängig* von Aufgabenträgern, die wir im Systemmodell durch Komponenten abbilden, beschreiben können. Das bedeutet unter anderem, dass beispielsweise eine komponentenbasierte Verhaltenssicht, in der Komponenteninteraktionen die Verhaltensbausteine darstellen, ungeeignet ist. Stattdessen wählen wir Entitäten und Prozessereignisse als die primären Modellbausteine, die *optional* Komponenten zugeordnet werden können.

Die Entitäten der Anwendungswelt sind „passive“ Objekte, die in Geschäftsprozessen bearbeitet werden und sowohl physische als auch ideelle Elemente darstellen können. Beispiele für Entitäten, wie Werkstücke und Konten, machen dies deutlich. Aufgrund des passiven Charakters von Entitäten, halten wir eine Modellierung fachlicher Entitäten in Form interagierender Objekte, wie dies in objektorientierten Ansätzen vorgeschlagen wird (was daran liegt, dass fachliche Entitäten auf dieselbe Art modelliert werden, wie Objekte, die das Softwaresystem bilden) für wenig intuitiv sowie für unnötig komplex und daher weniger geeignet.

Stattdessen modellieren wir Entitäten im Systemmodell als (Zustands-)Variablen, die in jedem Systemzustand mit Werten belegt sind. Durch Klassifikation von Entitäten legen wir die möglichen Zustandswerte sowie die zulässigen Zugriffsmög-

lichkeiten auf Entitätszustände in Form von Operationen fest. Diese Operationen bilden sozusagen eine kapselnde Schnittstelle von Entitäten, die einen kontrollierten Zugriff garantieren, ähnlich zu Objektmethoden in der Objektorientierung. Da es für die Beschreibung einer fachlichen Aufgabenstellung sinnvoll ist Entitätseigenschaften unabhängig von konkreten Datentypen/Repräsentationsformen angeben zu können, wählen wir das Systemmodell so, dass es ein geeignetes semantisches Modell für die algebraische Spezifikation (vergleiche gegebenenfalls [Wir90]) (von Entitätssorten) darstellt. Konkret verwenden wir Rechenstrukturen (siehe Definition 8.6) zur Beschreibung von Entitätssorten und zur Beschreibung des Systemzustandsraums.

Die Wahl der Systemmodellelemente zur Modellierung fachlicher Abläufe ergibt sich aus der Betrachtung bewährter Techniken zur Spezifikation von Geschäftsprozessen. Prominente Beispiele für diese Art von Technik sind ereignisgesteuerte Prozessketten (EPKs) [Schee98], eine Vielzahl von Petri-Netz-Varianten [ADO00] sowie die Aktivitätsdiagramme der UML [OMG01]. Allen diesen Techniken ist gemeinsam, dass sie Prozesse als eine Menge von Ereignissen und deren kausalen Abhängigkeiten modellieren. In unserem Systemmodell verwenden einen Prozessbegriff, der genau diesen konzeptuellen Kern der Techniken abdeckt. Ein Prozess besteht dabei im Wesentlichen aus einer Menge von Ereignissen sowie einer partiellen Ordnung auf der Ereignismenge, durch welche die kausalen Abhängigkeiten zwischen den Prozessereignissen erfasst werden. Die Partialität der Ordnung erlaubt es, im Unterschied zu Interleaving-Modellen, kausale Unabhängigkeit explizit zu erfassen (*true concurrency*). Auf die damit verbundenen Vorteile für unseren Ansatz, insbesondere in Verbindung mit der komponentenbasierten Perspektive unseres Systemmodells, gehen wir in Abschnitt 2.6.2 detailliert ein.

Die folgende Beschreibung der ganzheitlichen Perspektive beginnen wir mit der Modellierung fachlicher Entitäten und dem durch sie aufgespannten globalen Zustandsraum (Abschnitt 2.3.1) um anschließend Prozesse einzuführen (Abschnitt 2.3.2). Da, wie wir in Kapitel 4 sehen werden, die Spezifikation von Geschäftsprozessen der Formulierung von Reaktionseigenschaften des Anwendungssystems entspricht, geben wir in Abschnitt 2.3.3 das Reaktionsabbildung genannte Systemmodellelemente an, wodurch differenzierte Formen der Geschäftsprozessspezifikation möglich werden.

2.3.1 Entitäten und Zustandsraum der ganzheitlichen Perspektive

Wie bereits erwähnt, sind die Objekte der Anwendungswelt ein zentrales Element der fachlichen Aufgabenstellung einer Entwicklung. Werkstücke, Konten und Dokumente sind typische Beispiele für fachliche Entitäten, die deutlich machen, dass Entitäten „passiven" Einheiten (im Sinne bearbeiteter Objekte und nicht bearbeitender Subjekte) entsprechen.

Wir nennen fachliche Objekte *Entitäten*, einerseits in Anlehnung an das gleichnamige Konzept der Entity-Relationship-Modellierung, andererseits um Assoziationen mit Objekten im Sinne objektorientierter Programmiersprachen zu vermeiden. Dies liegt in dem passiven Charakter fachlicher Objekte begründet, der eher der Vorstellung einer Entität in der E/R-Modellierung entspricht, als einem nachrich-

tenempfangenden und –versendenden Objekt im Sinne objektorientierter Programmiersprachen.

Wie jedes reale Objekt ist auch eine Entität eine eindeutig identifizierbare Einheit. Daher modellieren wir eine Entität als Element einer Menge (von Identifikatoren).

Im allgemeinen interessieren wir uns für Entitäten, die unterschiedliche Zustände einnehmen können und deren Zustände durch „Bearbeitung“ verändert werden. So ändert sich beispielsweise der Zustand eines Leserkontos durch Rückbuchung eines Mediums. Im Systemmodell modellieren wir die Zustände, die Entitäten einnehmen können, durch Mengen von (Zustands-)Werten. Einen Zustand einer Entität modellieren wir durch eine Abbildung, die der Entität einen (Zustands-)Wert zuordnet, das heißt einer Abbildung der Form

Identifikator → (Zustands-)Wertmenge

Zustände eines Systems modellieren wir durch Abbildungen, die jeder Entität des Systems einen (Zustands-)Wert zuordnen:

Identifikatormenge → (Zustands-)Wertmenge

Die Wirkung von Bearbeitungsformen von Entitäten bilden wir durch Zustandsänderungen ab, das heißt durch Abbildungen der Form

Systemzustand → Systemzustand

Damit ist unser Modell ähnlich zu dem semantischen Modell imperativer Programmiersprachen: Entitäten entsprechen den Programmvariablen und Programmzustände entsprechen unseren Systemzuständen, modelliert durch Belegungsabbildungen der Variablen beziehungsweise der Entitäten (vergleiche zum Beispiel [LS84]).

Charakteristisch für eine Entität ist, welche Zustände sie einnehmen kann und welche Bearbeitungsformen auf sie angewendet werden können. Um für Entitäten diese Charakteristika festzulegen, wenden wir das Prinzip der Klassifikation an. Das bedeutet, dass wir jede Entität einer bestimmten Klasse/Sorte zuordnen und damit für jede Entität die Menge der möglichen Zustände und die möglichen Bearbeitungsformen festlegen. Beispielsweise sind für Entitäten der Sorte *Werkstück* die möglichen Zustände durch die Menge *{roh, geschliffen}* gegeben, eine mögliche Bearbeitungsform ist das *Schleifen* eines Werkstückes, was den Zustandsübergang von *roh* in *geschliffen* oder das Verharren im geschliffenen Zustand bedeutet.

Indem wir den Zustandsraum und die Bearbeitungsformen von Entitäten durch Klassifikation festlegen, ist unser Vorgehen ähnlich zur Objektorientierung, wo das Konzept der „Klasse“ zur Festlegung der Attribute und Methoden von Objekten dient (vergleiche zum Beispiel [Bud97]).

Die Klassifikation von Entitäten modellieren wir im Systemmodell durch Partitionierung der Entitätsmenge eines Systems in (disjunkte) Mengen, die sogenannten *Identifikatorsorten*. So unterscheiden wir zum Beispiel die Sorten der Werkstück- und der Leserkonten-Entitäten. Die Festlegung der Menge der Zustandswerte, die Entitäten einer bestimmten Klasse annehmen können, erfassen wir im Systemmodell dadurch, dass wir die Identifikatorsorten eines Systems als eine mit den Wertsorten des Systems indizierte Menge von Sorten angeben. Wir schreiben v und

id_v für eine Wertsorte v und die zugehörige Identifikatorsorte id_v. Auch hier besteht Ähnlichkeit zur Objektorientierung, wo mit jeder Klasse implizit auch eine Sorte der Identifikatoren von Objekten der jeweiligen Klasse eingeführt wird.

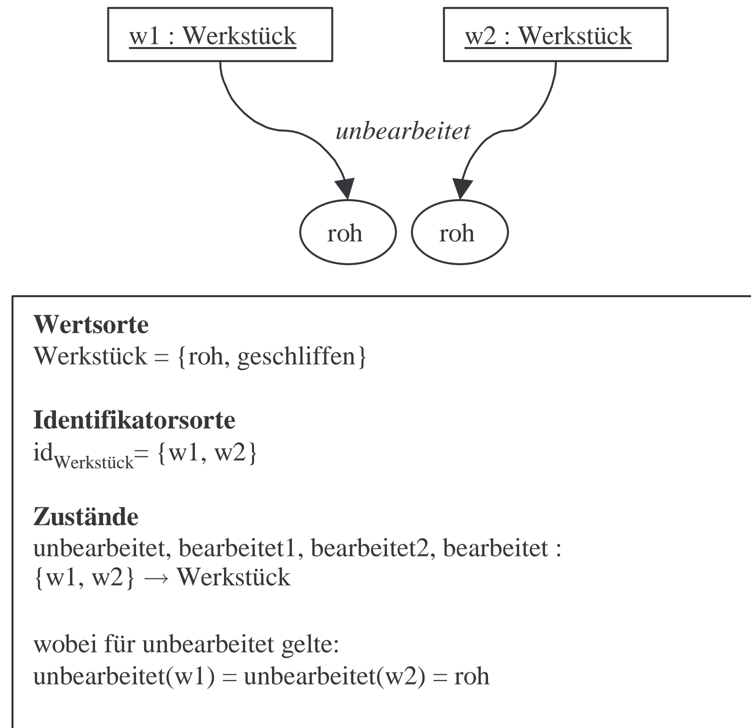


Abbildung 2.6: Entitäten, Sorten und Systemzustände

Das in Abbildung 2.6 illustrierte Beispiel veranschaulicht obige Systemmodellelemente anhand zweier Werkstück-Entitäten, *w1* und w2, den entsprechenden Sorten sowie anhand eines Systemzustandes, den wir *unbearbeitet* nennen und durch den beiden Entitäten jeweils der Zustandswert *roh* zugeordnet wird.

Mit den Wertsorten haben wir die Zustandsmengen der verschiedenen Arten von Entitäten im Systemmodell erfasst. Es stellt sich die Frage, in welcher Form wir diese Zustandsmengen beschreiben. Grundsätzlich können wir Wertmengen

- konkret angeben, in Form konkreter mathematischer Modelle/Datenstrukturen, beispielsweise auf der Grundlage von Mengen, Funktionen, Relationen oder Graphen, oder
- auf algebraische Weise angeben, indem wir eine Wertmenge mittels der auf ihren Elementen definierten Operationen und den zwischen den Operationen geltenden Gesetzmäßigkeiten charakterisieren.

Da die ganzheitliche Perspektive und damit die fachlichen Entitäten des Systemmodells nicht der Beschreibung (der Implementierung) des Softwaresystems, sondern der fachlichen Aufgabenstellung dienen, ist die Angabe konkreter Datenstrukturen weniger von Bedeutung. Vielmehr muss die Wirkung der fachlichen „Bearbeitungsformen“ von Entitäten angegeben werden. Dies ist mit der algebrai-

schen Spezifikation der Wertsorten möglich, indem fachliche Bearbeitungsformen durch die auf der jeweiligen Sorte definierten Operationen modelliert werden. Dementsprechend enthält unser Systemmodell Elemente, um die Wertsorten algebraisch charakterisieren zu können. Wir verwenden dabei die Konzepte der Signaturen und Rechenstrukturen, wie sie aus dem Gebiet der algebraischen Spezifikation bekannt sind (vergleiche hierzu gegebenenfalls Abschnitt 8.2):

- in einer Signatur, die wir Σ_{fe} (fachliche Entitäten) (vergleiche gegebenenfalls Definition 8.1) nennen, erfassen wir die Bezeichner der Identifikator- und der Wertsorten eines Systems, zusammen mit den Bezeichnern und Funktionalitäten ihrer charakteristischen Operationen.
- durch eine Σ_{fe}-Rechenstruktur, im Systemmodell mit *A* bezeichnet, beschreiben wir die Identifikator- und Wertmengen des Anwendungssystems, zusammen mit ihren Operationen.
- durch eine Menge C_f von Σ_{fe}-Rechenstrukturen erfassen wir die Menge der als zulässig betrachteten Rechenstrukturen des Anwendungssystems (in der ganzheitlichen Perspektive). Die Rechenstruktur A ist eine der durch C beschriebenen Alternativen, das heißt es muss gelten: $A \in C_f$ A. Neben A mit C_f auch die Menge der zulässigen Rechenstrukturen im Systemmodell zu erfassen, ermöglicht es uns, zwischen vollständigen und unvollständigen Spezifikationen zu unterscheiden. Auf diese Weise können wir beispielsweise unterscheiden zwischen dem Entwicklungsprodukt der exakten Spezifikation der Entitätsrechenstrukturen (siehe Definition 3.5), die als vollständige Spezifikation der (Eigenschaften der) Entitätsrechenstrukturen zu verstehen ist, und Produkten, die nur als partielle Spezifikationen zu interpretieren sind, beispielsweise um bestimmte Aspekte getrennt von anderen innerhalb eines Produktes überschaubar zu erfassen (siehe Kapitel 3).

Die oben beschriebene 1:1 Zuordnung von Wert- und Identifikatorsorten in der Entitätssignatur Σ_{fe} und den Σ_{fe}-Rechenstrukturen eines Systems erlaubt uns die Modellierung von Referenzen zwischen Entitäten. Damit können wir so wichtige Modellierungskonzepte wie *Relationen* der Entity-Relationship-Modellierung und *Assoziationen* objektbasierter Modellierung (vergleiche etwa UML-Klassendiagramme [OMG01]) im Systemmodell abbilden (siehe auch Abschnitt 2.4.1).

Definition 2.6 (fachliche Wertsorten, Entitätssignatur, Rechenstruktur-Klasse, und Rechenstruktur)

- Die endliche Menge der *fachlichen Wertsorten* bezeichnen wir mit *V*.
- Die (Bezeichner der) Wert- und Identifikatorsorten eines Systems, zusammen mit ihren charakteristischen Operationen, erfassen wir in der *fachlichen Entitätssignatur* Σ_{fe} . Σ_{fe} ist eine Entitätssignatur bezüglich V (gemäss Definition 8.8 (Entitätssignaturen)). Wir fordern

  § $\Sigma_{BOOL} \subseteq \Sigma_{fe}$ und

  § $Act \in sorts(\Sigma_{fe})$ und $Act \notin V$.

  (bezüglich Σ_{BOOL} siehe Definition 8.7)
- C_f steht für die *Klasse der fachlichen Rechenstrukturen*, wobei

  $C_f \subseteq Gen(\Sigma_{fe})$[12] und

  $C_f \big|_{\Sigma_{BOOL}} = C_{BOOL}$

  gelte[13]. Zudem sei jede Rechenstruktur $c \in C_f$ *total*.
- Mit *A* bezeichnen wir die *fachliche Rechenstruktur*, wobei $A \in C_f$ gelte.

Für den weiteren Gebrauch definieren wir die Mengen

$$A_V =_{def} \bigcup_{s \in V} A_s$$

und

$$A_{ID} =_{def} \bigcup_{s \in V} A_{id_s}$$

der Werte- beziehungsweise Identifikator-Trägermengen.

□

Mit obiger Definition steht uns der abstrakte Datentyp C_{BOOL} (aus Definition 8.7) in jedem System zur Verfügung. Wir benötigen dies, um in Abschnitt 2.3.2 Zustandsprädikate als mögliche Markierungen von Prozessereignissen definieren zu können. Ebenso lassen wir Elemente der Trägermenge A_{Act} als Ereignismarkierungen zu (vergleiche Definition 2.11). Die (im Systemmodell) ausgezeichnet Sorte *Act* steht für Aktionsbezeichner. Wir modellieren Aktionsbezeichner als eine Trägermenge der fachlichen Rechenstruktur *A*, da wir Aktionsbezeichner häufig in Abhängigkeit von (Trägermengen von) Wert- und Identifikatorsorten definieren. Ein Beispiel ist die Aktionsbezeichnung *vormerkungswunsch(m,l)*. Ein damit markiertes Prozessereignis steht für die Äußerung eines Vormerkungswunsches für das Medium *m* und den Leser *l*. Die Funktion

---

[12] für jede Signatur Σ steht $Gen(\Sigma)$ für die Menge der Σ-Rechenstrukturen (siehe Definition 8.6)

[13] bezüglich C_{BOOL} siehe Definition 8.7

vormerkungswunsch : $id_{Medium} \times id_{Leser} \rightarrow Act$

ist ein Konstruktor für Aktionsbezeichner (siehe Beispiel 4.1).

C_f stellt die Klasse der zulässigen Rechenstrukturen dar. A ist die konkrete Rechenstruktur eines Systems. Ähnlich zu abstrakten Datentypen (die gemäss Definition 8.6 Isomorphie-Klassen von Rechenstrukturen darstellen) haben wir mit C_f eine repräsentationsunabhängige Sicht der Entitätseigenschaften eines Systems. Diese abstrakte Sicht erlaubt uns die klare semantische Unterscheidung von unvollständigen und vollständigen Datenspezifikationen (siehe Kapitel 3). Die Rechenstruktur A repräsentiert die konkreten Entitätssorten und -Operationen eines Systems und ist eine der in C_f enthaltenen (Implementierungs-)Alternativen.

Folgende Betrachtungen motivieren die mit Definition 2.6 getroffenen Entwurfsentscheidungen für das Systemmodell:

- **Rechenstrukturen.** Rechenstrukturen stellen die semantische Domäne eigenschaftsorientierter, algebraischer Spezifikationen dar. Die Eignung des Systemmodells für diese Art der Spezifikation erscheint uns wichtig, da wir durch die algebraische Spezifikation eine repräsentationsunabhängige Sicht auf fachliche Entitäten gewinnen. Dadurch können wir Überspezifikation vermeiden. Die Beschreibung *fachlicher* Elemente bleibt dadurch frei von (der unfreiwilligen Vorwegnahme von) Implementierungsdetails. In [ZJ97] wird diese Thematik unter dem Begriff *implementation bias* diskutiert und algebraische Techniken werden als ein Ansatz zur Vermeidung von *implementation bias* erwähnt.

  Vor dem Hintergrund der weiten Verbreitung objektorientierter Implementierungsplattformen bietet die repräsentationsunabhängige Datensicht algebraischer Spezifikationen zudem den Vorteil eines nahtlosen Übergangs von der Spezifikation zur Implementierung. Wie in Abschnitt 3.4.2 anhand eines Beispiels illustriert, können wir die (Gleichungs-)Axiome algebraischer Spezifikationen als die geforderten Eigenschaften und Testfälle der (öffentlichen) Objektmethoden einer Klasse (im Sinne der Objektorientierung) verstehen.

  Algebraische Spezifikationstechniken sind im allgemeinen relativ abstrakt, da die Spezifikationskonzepte allgemein gehalten sind. Im Unterschied dazu, umfassen Ansätze wie die Entity-Relationship-Modellierung Konzepte, die auf die Datenmodellierung von Informationssystemen zugeschnitten sind. In [Het95] wird die Entity-Relationship-Modellierung durch Abbildung auf Σ-Rechenstrukturen semantisch fundiert. In [Bre91] wird die Semantik einer algebraischen Spezifikationssprache für objektorientierte Klassen sowie einer objektorientierten, zustandsbasierten Programmiersprache durch Σ-Algebren beschrieben. Diese beiden Beispiele unterstreichen die Eignung unseres Systemmodells für verbreitete Datenmodellierungsansätze.

- **Termerzeugtheit.** In C_f und A beziehen wir uns nicht auf beliebige Σ_{fe}-Algebren, sondern beschränken uns auf erreichbare/termerzeugte Algebren, das heißt auf Σ_{fe}-Rechenstrukturen. Damit sind nur endliche, dis-

krete Wertemengen erfassbar. Für die Modellierung von Softwaresystemen ist diese Einschränkung notwendig, da dort alle Objekte endlich darstellbar sein müssen. Ebenso ist dieses eingeschränkte Modell im allgemeinen für die Modellierung der Anwendungssysteme sinnvoll, im Rahmen derer Informationssysteme typischerweise zum Einsatz kommen. Im Zusammenhang mit hybriden Anwendungssystemen[14], die häufig den Kontext für (eingebettete) Steuerungssysteme bilden, kann eine Anwendungssystem-Sicht mit diskreten Wertemengen ein sinnvolles Bindeglied zwischen einer hybriden Anwendungssystem-Sicht und der notwendigerweise diskreten Softwaresystem-Sicht darstellen. Durch die diskrete Anwendungssystem-Sicht können wir die fachliche Abstraktion von kontinuierlichen zu diskreten Wertemengen dokumentieren.

Ein weiterer, eher verifikationstechnischer Vorteil der Termerzeugtheit ist, dass wir Abstraktionsabbildungen zwischen den Rechenstrukturen (verschiedener Detaillierungsebenen oder unterschiedlicher Perspektiven) induktiv definieren können, was zu klaren und leicht operationalisierbaren Definitionen der Abstraktionsabbildungen führt. Unter der operationalisierten Form einer Abstraktionsabbildung können wir beispielsweise Algorithmen verstehen, welche, an der Schnittstelle des Softwaresystems, die systeminternen technischen Daten in fachliche Daten transformieren (Abstraktion von technischen Details). Zudem können wir bei Beweistechniken auf das Prinzip der strukturellen Induktion zurückgreifen.

- **Totalität.** Indem wir die Totalität von A und der Elemente von C_f fordern, schränken wir die Ausdrucksmächtigkeit bewusst ein. Dadurch sparen wir explizit die Problematik aus, die sich ergibt, wenn wir Prozessereignisse mit undefinierten Werten markieren (vergleiche Abschnitt 2.3.2) und auf diese Weise „nichtterminierende" Ereignisse von Prozessen zu behandeln hätten.
- **Wert- und Identifikator-Sorten.** Mit der Forderung, dass Σ_{fe} eine Entitätssignatur (gemäss Definition 8.8) bezüglich einer Menge V von Sorten ist, unterscheiden wir zwischen den *Wertsorten* $s \in V$ und den ihnen zugeordneten *Identifikatorsorten* id_s . Rekursion vermeiden wir, indem wir $V \cap ID = \emptyset$ fordern. Für die folgende Definition der Begriffe des Systemzustandsraums nutzen wir diese Eigenschaften der Signatur Σ_{fe}. Zudem gehen wir in Abschnitt 2.4.1 auf die Abbildung von Assoziationen, im Sinne der objektorientierten Modellierung, beziehungsweise von Relationen, im Sinne einer Entity-Relationship-Modellierung, mittels Identifikatorsorten ein.

Wie eingangs des Kapitels motiviert, sind die fachlichen Entitäten des Anwendungssystems eindeutig identifizierbare Einheiten, die wir als Elemente der Identifikatorsorten im Systemmodell erfassen:

[14] (vergleiche [Stau01]): Systeme, die auch nicht-diskrete Wertemengen, wie beispielsweise die reellen Zahlen, umfassen und die einen nicht-diskreten Zeitbegriff verwenden.

Definition 2.7 (Menge der Entitäten)

Die endliche Menge der Entitäten eines Systems nennen wir *E* und fordern, in Abhängigkeit von der Entitätsrechenstruktur A:

$E \subseteq A_{ID}$

□

Entitäten bilden im Systemmodell die Grundlage für den fachlichen Zustandsbegriff. Analog zu den Objekten der Anwendungswelt, deren Zustände den Zustand des Anwendungssystems bestimmen, beispielsweise bestimmen die Zustände der Leserkonten und Medien den Zustand einer Bibliothek, ergibt sich auch in unserem Systemmodell ein fachlicher (Daten-)Zustand aus den Zuständen der Entitäten des Systems. Wie erwähnt, modellieren wir daher Systemzustände als sortenkonforme Belegungen der Entitäten. Neben dem Zustandsbegriff sind auch die Operationen auf Zuständen in der ganzheitlichen Perspektive von Bedeutung, da etwa das Verwalten fachlicher Information die Änderung von Entitätszuständen bedeutet.

In den beiden folgenden Definitionen fassen wir die Sorte der Systemzustände und die Operationen auf Systemzuständen in Form einer Algebra, die wir die *Zustandsalgebra* nennen, zusammen. Durch diese Zusammenfassung in einer Algebra, können wir in Abschnitt 2.6.1 das Konzept des Homomorphismus zwischen Algebren verwenden, um eine Realisierungsbeziehung zwischen dem Systemzustand der ganzheitlichen und der komponentenbasierten Perspektive herzustellen.

Für einen systematischen Entwicklungsprozess ist es wichtig, dass unser Systemmodell die systematische Festlegung von Operationen auf dem Zustandsraum unterstützt. Wir verfolgen hier einen Ansatz, ähnlich der objektorientierten Spezifikation und Programmierung, wo durch die Klasse, der ein Objekt angehört, die zulässigen Operationen auf dem durch das Objekt gekapselten Zustandsraum festgelegt sind (die Anwendung von Klassenmethoden auf ein bestimmtes Objekt entspricht diesen Operationen). Wir übertragen dies auf unseren Ansatz gemäss folgender Vorstellung:

Wie oben erwähnt, zeichnen sich unterschiedliche Arten von Entitäten darin aus, welche Zustände eine Entität einnehmen kann und welche Bearbeitungsformen, das heißt Operationen, auf diese Zustände anwendbar sind. Im Systemmodell erfüllen wir die Anforderung, dass Entitäten nur ihrer Klassifikation entsprechende Zustände einnehmen können, durch eine geeignete Definition der Menge der Systemzustände (siehe Definition der Trägermenge der Sorte *State* für die Zustandsalgebra *FS* in unten gegebener Definition 2.9):

Von jedem Systemzustand *s* fordern wir, dass für eine Entität *e* der Sorte id_v, wobei *v* eine Wertsorte sei, gelte:

$s(e) \in A_v$.

Die (in der Entitätsrechenstruktur A eines Systems angegebenen) Operationen auf *Wert*sorten stellen damit Operationen auf den Zustands*werten* von Entitäten dar. Um, im Rahmen einer Entwicklung, durch Definition dieser Operationen auch die Zugriffsmöglichkeiten auf Entitätszuständen festzulegen, leiten wir im Systemmodell die Menge der möglichen Operationen auf Systemzuständen aus den Operatio-

nen auf Wertsorten ab. Sei beispielsweise *w1* eine als Werkstück klassifizierte Entität, das heißt

$w1 \in id_{Werkstück}$

und sei für Werkstücke die Bearbeitungsform *schleifen* gegeben, das heißt

schleifen : Werkstück $\rightarrow$ Werkstück

sei eine Operation der Entitätsrechenstruktur A. Dann leiten wir daraus eine Operation

w1.schleifen : Systemzustand $\rightarrow$ Systemzustand

ab, für die, für alle Systemzustände s, gilt:

w1.schleifen(s) = s[w1/schleifen(s(w1))].

Das heißt, die Operation *w1.schleifen* verändert den Zustand der Entität *w1* gemäss der Operation *schleifen*. Beispiel 2.2 illustriert diese Form der Ableitung noch ausführlicher.

In Definition 2.11 legen wir für das Systemmodell fest, dass nur solche zustandsbezogenen Prozessereignisse zulässig sind, die der Anwendung einer Operation entsprechen, die auf oben beschriebene Weise aus einer Operation auf einer entsprechenden Wertsorte abgeleitet ist. Auf diese Weise bestimmt die eigenschaftsorientierte Spezifikation von Wertsorten die Zugriffsmöglichkeiten auf Entitäten, genauer gesagt auf Entitätszustände. Sei beispielsweise auf Werkstücken außer der Operation *schleifen*, mit

schleifen(roh) = geschliffen

und

schleifen(geschliffen) = geschliffen

(wobei $A_{Werkstück}$ = {roh, geschliffen} gelte)

keine weitere Operation (in der Entitätsrechenstruktur A) definiert, dann gibt es kein Prozessereignis, dass für die Zustandsänderung von Werkstücken von *geschliffen* nach *roh* steht (was einer aus fachlicher Sicht sinnvollen Einschränkung entspricht). Dieses Beispiel macht deutlich, dass unser Ansatz eine *systematische* Festlegung von Zustandsoperationen darstellt, da mit der Klassifikation der Entitäten zusammen mit der Spezifikation der Wertsorten auch die Zugriffsmöglichkeiten auf die Zustände von Entitäten festgelegt sind. Die *abgeleiteten* Zugriffsoperationen bilden somit die Schnittstelle von (Zuständen von) Entitäten, wie in Abbildung 2.7 graphisch dargestellt.

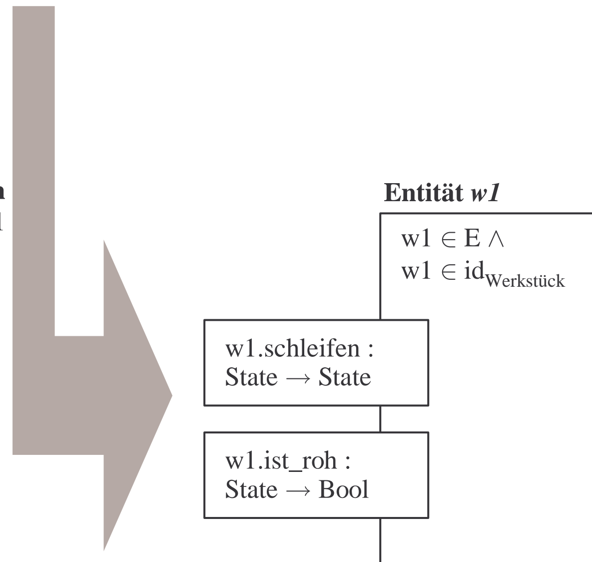


Abbildung 2.7: Ableitung von Zugriffsoperationen auf (Zustände von) Entitäten

Durch die Ableitung der zulässigen Operationen auf Systemzuständen aus den Operationen der Entitätsrechenstruktur, erreichen wir die Kapselung von Entitätszuständen, ähnlich zur Kapselung von Objektzuständen (via Zugriffsmethoden), wie wir sie aus der Objektorientierung kennen. Im Unterschied zu objektorientierten Ansätzen, in denen (auch fachliche) Objekte als durch Nachrichtenaustausch interagierende Elemente modelliert werden (vergleiche beispielsweise [Bre98, CAB+94, DW99, Rum96]), reicht für unseren Ansatz die relativ einfache Modellvorstellung der Entität als „belegbare Variable“ aus. So vermeiden wir zum einen unnötige Komplexität des Modells. Zum zweiten entspricht das Modell dem passiven Charakter fachlicher Entitäten, im Sinne *bearbeiteter* Objekte der Anwendungswelt.

Ähnlich zu oben beschriebenem Zusammenhang zwischen Operationen auf Wertsorten und Operationen auf Systemzuständen, wird in [Bre91] der Zusammenhang zwischen einer algebraischen, objektorientierten Spezifikationssprache mit einer zustandsbasierten, objektorientierten Programmiersprache hergestellt.

Wir geben nun die präzisen Definitionen der fachlichen Zustandssignatur sowie der fachlichen Zustandsalgebra an.

Die Signatur der fachlichen Zustandsalgebra umfasst neben der ausgezeichneten Sorte *State* auch die Sorten der Entitätssignatur sowie Bezeichner und Funktionalität der, wie oben beschrieben, abgeleiteten Operationen auf Systemzuständen. Die Sorten der Entitätssignatur benötigen wir an dieser Stelle, da die abgeleiteten Operationen (im allgemeinen) Parameter dieser Sorten besitzen.

Definition 2.8 (fachliche Zustandssignatur)

Die fachliche Zustandssignatur nennen wir Σ_{fs}. Für Σ_{fs} gelte, in Abhängigkeit von V, Σ_{fe} und der Entitätsmenge E:

i. $sorts(\Sigma_{fs}) =_{def} \{State\} \cup sorts(\Sigma_{fe})$

ii. $opns(\Sigma_{fs}) \subseteq E \times opns(\Sigma_{fe}) \times \{sel, kons\}$, mit

$$opns(\Sigma_{fs}) =_{def} \{(e,f,x) \in E \times opns(\Sigma_{fe}) \times \{sel, kons\} \mid \exists s \in V. \\ (e \in id_s) \wedge (selektor \vee konstruktor)\}$$

wobei

$$selektor =_{def} \left(x = sel \wedge s = ft\left(\Pi_1\left(type_{\Sigma_{fe}}(f)\right)\right)\right)$$

$$konstruktor =_{def} \left(x = kons \wedge s = ft\left(\Pi_1\left(type_{\Sigma_{fe}}(f)\right)\right) = \Pi_2\left(type_{\Sigma_{fe}}(f)\right)\right) \vee \\ \left(x = kons \wedge type_{\Sigma_{fe}}(f) = (\varepsilon, s)\right)$$

iii. für $type_{\Sigma_{fs}}$ gelte:

$$\forall (e,f,z) \in opns(\Sigma_{fs}), x, y. \\ type_{\Sigma_{fe}}(f) = (x,y) \Rightarrow \\ \left((z = kons \Rightarrow type_{\Sigma_{fs}}((e,f,z)) = (State \,\&\, rt(x), State)\right) \wedge \\ \left(z = sel \Rightarrow type_{\Sigma_{fs}}((e,f,z)) = (State \,\&\, rt(x), y)\right)\right)$$

Wir schreiben auch

- *e.f* für (e, f, sel) $\in opns(\Sigma_{fe})$ und
- *e.f** für (e, f, kons) $\in opns(\Sigma_{fe})$.

□

Die Elemente eines Funktionsbezeichners *(e, f, z)* haben folgende Bedeutung:

- *e* steht für die Entität, auf deren Zustand die Funktion *f* aus Σ_{fe} angewandt wird
- *z* klassifiziert die Funktion als Selektor- (sel) oder Konstruktorfunktion (kons). Diese Klassifikation legt fest, ob es sich um „die Abfrage einer Zustandseigenschaft" handelt, oder um eine Zustandsänderung. Damit bestimmt die Klassifikation als Selektor- beziehungsweise Konstruktorfunktion auch die durch $type_{\Sigma_{fs}}$ gegebene Funktionalität von (e, f, z)

Eigenschaft ii. bedeutet, dass wir für die Ableitung von Zustandsoperationen nur

$f \in opns(\Sigma_{fe})$ mit $f : s, ... \rightarrow s'$, $s \in V$

betrachten, das heißt nur Funktionen, deren erster Parameter eine Wertsorte ist. Diese syntaktische Festlegung hat zwei Gründe:

- Zum einen können nur aus den Funktionen Zugriffsfunktionen auf Entitäten abgeleitet werden, die (mindestens) einen Parameter einer Wertsorte besitzen. Dies liegt daran, dass wir unter „Zugriff“ auf eine Entität *e* gemäss *f* verstehen, dass sich der Zustand von *e* so ändert, wie es der Anwendung von *f* auf den aktuellen Zustand von *e* entspricht (vergleiche Definition 2.9). Da der Zustand einer Entität immer Element der zugehörigen Wertsorte ist, muss *f* einen Parameter dieser Wertsorte besitzen.
- Zum zweiten wird auf diese Weise, im Falle mehrerer Parameter der Sorte *s*, in der abgeleiteten Funktion *e.f* beziehungsweise *e.f** eindeutig festgelegt, welcher Parameter mit dem Zustand der Entität belegt wird (siehe Definition 2.9).

Des weiteren legen wir mit ii. fest, dass aus Funktionen

$f \in opns(\Sigma_{fe})$ mit $f : s, ... \rightarrow s,\ s \in V$

ambivalent sowohl Selektor- als auch Konstruktorfunktionen abgeleitet werden. Aus Funktionen

$f : s, ... \rightarrow s'$, mit $s \neq s'$

leiten wir nur Selektorfunktionen ab.

Mit Eigenschaft iii. legen wir die Funktionalität einer abgeleiteten Funktion, entsprechend der Klassifikation als Selektor- beziehungsweise Konstruktorfunktion, fest:

- $e.f : State, s1, ..., sn \rightarrow sy$, falls $f : s, s1, ..., sn \rightarrow sy$ in Σ_{fe}, und
- $e.f^* : State, s1, ..., sn \rightarrow State$, falls $f : s, s1, ..., sn \rightarrow s$ in Σ_{fe}.

Eine Selektor-Funktion *e.f* interpretieren wir in der Zustandsalgebra *FS* (siehe Definition 2.9) als „Anwendung der Funktion *f* auf den Zustand der Entität *e*“, wobei *e.f* einen Wert liefert, der Auskunft über den Zustand von *e* gibt.

Eine Konstruktor-Funktion *e.f** interpretieren wir in der Zustandsalgebra ebenfalls als eine Anwendung der Funktion *f* auf *e*, wobei *f* den Zustand von *e* verändert.

Funktionsbezeichner der Form *e.f* werden in [Hoa72] als *compound identifier* bezeichnet und haben eine ähnliche Bedeutung zu unserer Anwendung, die wir mit folgender Definition der Zustandsalgebra festlegen:

Definition 2.9 (fachliche Zustandsalgebra)

Die Eigenschaften der fachlichen Zustandsalgebra $FS \in Alg(\Sigma_{fs})$ definieren wir bezüglich der fachlichen Entitätsrechenstruktur A aus Definition 2.6:

i. $FS_s =_{def} A_s$, für alle $s \in sorts(\Sigma_{fe})$

ii. $FS_{State} =_{def} \{\sigma : E \rightarrow A_V \mid \forall e \in E, s \in V.\ e \in A_{id_s} \Rightarrow \sigma(e) \in A_s\}$

iii. für alle $e.f \in opns(\Sigma_{fs})$ mit $e.f : State, s1, ..., sn \rightarrow s$ gelte:

$\forall a1 \in FS_{s1}, ..., an \in FS_{sn}, \sigma \in FS_{State}.$
$e.f^{FS}(\sigma, a1, ..., an) = f^{A}(\sigma(e), a1, ..., an)$

iv. für alle e.f* $\in$ opns(Σ_{fs}) mit e.f : State, s1, ..., sn $\to$ State gelte:

$\forall a1 \in FS_{s1}, ..., an \in FS_{sn}, \sigma \in FS_{State}.$
$e.f^{*FS}(\sigma, a1, ..., an) = \sigma\left[e / f^{A}(\sigma(e), a1, ..., an)\right]$

□

In Abschnitt 2.4.2 gehen wir auf Eigenschaften der abgeleiteten Operationen auf Zuständen noch einmal näher ein. Das folgende Beispiel illustriert die syntaktischen Aspekte des Ableitungsansatzes.

Beispiel 2.2 (Ableitung von Operationen einer Zustandsalgebra)

Sei

sorts Medium, id_{Leser}, Status;
mkons : Medium;
ausleihe : Medium $\times id_{Leser} \to$ Medium;
entleiher : Medium $\to id_{Leser}$;
status : Medium $\to$ Status;

ein Ausschnitt der Entitätssignatur eines Systems *sys*, welche die Sorte *Medium* und einige ihrer charakteristischen Operationen umfasst. Sei weiterhin

$m \in A_{id_{Medium}}$ und $m \in E$.

Dementsprechend umfasst die Zustandssignatur $(\Sigma_{fs})_{sys}$ folgende Operationen:

m.mkons* : State $\to$ State;
m.ausleihe* : State $\times id_{Leser} \to$ State;
m.ausleihe : State $\times id_{Leser} \to$ Medium;
m.entleiher : State $\to id_{Leser}$;
m.status : State $\to$ Status;

□

Neben der Festlegung der zustandsbezogenen Operationen, bleibt im Systemmodell noch der Initialzustand des Systems festzulegen.

Definition 2.10 (Initialzustand)

Den Initialzustand des Systems bezeichnen wir mit *init_state*, wobei gelte

init_state $\in$ FS_{State}.

□

2.3.2 Prozesse

Neben den Objekten der Anwendungswelt, den fachlichen Entitäten, sind auch die Abläufe der Anwendungswelt Teil der fachlichen Aufgabenstellung einer Entwicklung. Für Informationssysteme sind dies die Geschäftsprozesse von Unternehmen und Organisationen im allgemeinen.

Dementsprechend beschreiben wir das Verhalten des Anwendungssystems in der ganzheitlichen Perspektive des Systemmodells durch Prozesse. Dabei stellt sich die Frage, wie wir Prozesse modellieren ? Wir haben uns für ein Modell entschieden, dass ähnlich ist zu den *labelled partial orders* aus [Pra86], den *action structures* aus [Bro89] und den *configuration structures* (siehe etwa [GG98]). Ein Prozess besteht dabei aus

- einer Menge von *Ereignissen*,
- einer *partiellen Ordnung* auf der Ereignismenge sowie aus
- einer *Markierungsabbildung*, die jedem Prozessereignis eine Markierung zuordnet.

Durch die Partialordnung eines Prozesses erfassen wir die kausalen Abhängigkeiten zwischen den Prozessereignissen. Jedes Prozessereignis steht für eine bestimmte fachliche Aktivität, etwa der Bearbeitung einer Entität. Die fachlichen Aktivitäten modellieren wir durch die Menge der Ereignismarkierungen einer ganzheitlichen Perspektive (vergleiche gegebenenfalls Definition 2.11). Mit Hilfe der Markierungsabbildung eines Prozesses legen wir fest, für welche Aktivität ein bestimmtes Prozessereignis steht.

Im Systemmodell unterscheiden wir drei Arten von Ereignismarkierungen, entsprechend den zentralen Ereignisformen in Geschäftsprozessen:

- Um die Bearbeitung von Entitäten zu erfassen, markieren wir Ereignisse mit Abbildungen der Form

  Systemzustand $\rightarrow$ Systemzustand

- Da der Verlauf fachlicher Abläufe vielfach von Entitätszuständen abhängig ist, beispielsweise folgt auf die Äußerung eines Entleihwunsches nur dann die Ausleihe des gewünschten Mediums, wenn das Medium verfügbar ist (sonst wird der aktuelle Entleiher zur Rückgabe aufgefordert), stellt die Abfrage/Beobachtung von Zuständen eine typische fachliche Aktivität dar. Wir modellieren dies durch Zustandsprädikate als Ereignismarkierungen, das heißt durch Abbildungen der Form

  Systemzustand $\rightarrow$ Bool

- Darüber hinaus gibt es noch Aktivitäten, die weder Zustandsänderungen noch Zustandsabfragen darstellen und daher, neben der Bedeutung ihres Bezeichners, innerhalb der ganzheitlichen Perspektive nicht näher interpretiert werden. Ein Beispiel ist die Rückgabeaufforderungen eines Lesers per E-Mail. Diese Aktivität wird erst in der komponentenbasierten Perspektive durch ein entsprechendes Interaktionsmuster zwischen Komponenten interpretiert. Für diese allgemeine Form von Aktivität führen wir die Menge der sogenannten *Aktionen*, als Teilmenge der Ereignismarkierungen eines Systems, ein. Wir modellieren die Aktionsbezeichner durch die ausgezeichnete Sorte *Act* der Entitätsrechenstruktur (vergleiche Definition 2.6).

Der oben skizzierte Prozessbegriff stellt den konzeptuellen Kern gängiger und bewährter Techniken der Geschäftsprozessmodellierung dar. Prominente Beispiele für diese Art von Techniken sind

- ereignisgesteuerte Prozessketten (EPKs) [Schee98],
- eine Vielzahl von Petri-Netz-Varianten [ADO00, JVW00] sowie
- die Aktivitätsdiagramme der UML [OMG01, Nüt98].

Die Angemessenheit, eine Partialordnung als Grundstruktur von Prozessen zu wählen, zeigt sich etwa bei der Betrachtung der Grundstruktur der Notationen aller genannten Techniken. Dies sind gerichtete Graphen, deren Knoten Ereignisse und deren Kanten die kausalen Abhängigkeiten zwischen Ereignissen, die Partialordnung, repräsentieren. Auch die Tatsache, dass in [LSW97] ereignisgesteuerten Prozessketten durch Abbildung auf Petri-Netze eine Bedeutung gegeben wird und in [Win87b] wiederum für Petri-Netze eine Semantik mittels *event-structures* definiert wird, die in [GG98] auf *configuration-structures* zurückgeführt werden, unterstreicht die Ädaquatheit unserer Modellierung.

Neben der konzeptuellen Nähe zu den Techniken der Geschäftsprozessmodellierung, spricht für ein Partialordnungsmodell, dass diese Modellart in der Informatik eine wichtige Rolle für die Charakterisierung verteilter Systeme spielt, wie das folgende Zitat zeigt:

> *“The heart of the approach is a notion of partial string derived from the view of a string as a linearly ordered multiset by relaxing the linearity constraint, thereby permitting partially ordered multisets or* pomsets. *[...]*
>
> *The general benefits of the approach are that it is conceptually straightforward, involves fewer artificial constructs than many competing models of concurrency, yet is applicable to a considerably wider range of types of systems [...].”* [**Pra86**]

Siehe auch [**Pra86**], wo die in obigem Zitat gemachten Aussagen anhand von Beispielen und einem Vergleich des Partialordnungsansatzes mit Interleavingmodellen belegt werden.

Ziel der Beschreibung von Geschäftsprozessen ist es, kausale Abhängigkeiten zwischen Ereignissen anzugeben und nicht die Menge aller beobachtbaren Folgen von Ereignissen. Da im allgemeinen auch kausal unabhängige Ereignisse in Geschäftsprozessen enthalten sind, würde eine Interleaving-Modell eine „künstliche“ Totalisierung der Kausalordnung einführen und stellt daher unserer Meinung nach kein

intuitives Modell für fachliche Abläufe dar. In dieser Hinsicht unterscheidet sich unser Ansatz etwa von [Web91] und [LT89], wo zur Beschreibung einer Aufgabenstellung Spurmodelle verwendet werden.

Das in Abbildung 2.8 skizzierte Beispiel eines Überweisungsvorgangs veranschaulicht die Angemessenheit eines Partialordnungsmodells für fachliche Abläufe.

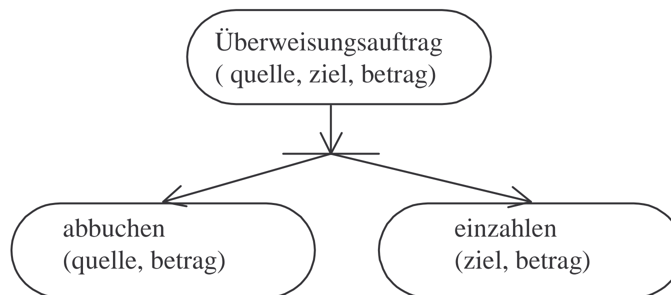


Abbildung 2.8: (Graph eines) Überweisungsvorgangs

Ein weiterer, methodisch bedeutsamer Aspekt wird in [GG98] angesprochen:

> *"We study an operator for refinement of actions to be used in the design of concurrent systems. Actions on a given level of abstraction are replaced by more complicated processes on a lower level. This is done in such a way that the behavior of the refined system may be inferred compositionally from the behavior of the original system and from the behavior of the processes substituted for actions. We recall that interleaving models of concurrent systems are not suited for defining such an operator in its general form. Instead, we define this operator on several causality based, event oriented models [...]."* [**GG98**]

Das Zitat weist auf einen für die fachliche Verhaltensmodellierung wichtigen Vorteil von Partialordnungs- gegenüber Interleaving-Modellen hin:

Für Partialordnungsmodelle stehen kompositionale Verfeinerungsbegriffe zur Verfügung, wodurch eine hierarchische Verhaltensstrukturierung und damit etwa eine Entwicklung gemäss der Idee der schrittweisen Verfeinerung im Sinne von [Wir71] ermöglicht wird.

Ein weiterer Vorteil des Partialordnungsmodells ist, dass es durch die echte/explizite Nebenläufigkeit/Unabhängigkeit (durch Partialität der Kausalrelation) hilft, Überspezifikation zu vermeiden. Das ist für die ganzheitliche Perspektive von besonderer Bedeutung, da diese zur Erfassung der fachlichen *Aufgabenstellung* dient, und somit frei von Realisierungsaspekten zu sein hat. Insbesondere im Hinblick darauf, dass unsere komponentenbasierte Perspektive, in der wir die Realisierung einer Aufgabenstellung angeben, auch ein Modell mit expliziter Nebenläufigkeit (und kein reines Interleaving-Modell) ist, müssen alle fachlich zulässigen Freiheitsgrade (für Nebenläufigkeit) erhalten bleiben. Wir gehen in Abschnitt 2.6.2 detailliert darauf ein.

In den folgenden beiden Definitionen legen wir Eigenschaften der Menge der Ereignismarkierungen und der Prozesse eines Systems fest.

Für die Charakterisierung der Ereignismarkierungen nehmen wir Bezug auf die Entitätsalgebra *A* und die Zustandsalgebra *FS*. Während die Trägermenge A_{Act} zur Markierung von Aktionsereignissen dient, leiten wir aus den Operationen der Zustandsalgebra *FS* die als Markierung verwendeten Zustandsprädikate und Zustandsabbildungen ab:

Definition 2.11 (Ereignismarkierungen, Zustandsrelationen)

Die Menge der Markierungen *M* von Prozessereignissen definieren wir in Abhängigkeit von der Signatur Σ_{fs} und der Zustandsalgebra FS. Wir unterscheiden die drei folgenden Arten von Ereignismarkierungen:

- $T \subseteq FS_{State} \rightarrow FS_{State}$,

  $T =_{def} \{g : FS_{State} \rightarrow FS_{State} | q(g)\}$ und

- $Z \subseteq FS_{State} \rightarrow FS_{Bool}$,

  $Z =_{def} \{g : FS_{State} \rightarrow FS_{Bool} | q(g)\}$

  mit

$$q(g) \Leftrightarrow_{def} \exists f \in opns(\Sigma_{fs}), s_1, \dots, s_n \in sorts(\Sigma_{fs}). \exists a_1 \in FS_{s_1}, \dots, a_n \in FS_{s_n}. \forall \sigma \in FS_{State}. g(\sigma) = f^{FS}(\sigma, a_1, \dots, a_n)$$

- A_{Act}.

Die Menge der Ereignismarkierungen ist definiert durch

$$M =_{def} (T \cup Z \cup A_{Act}).$$

Durch jedes $t \in T$ wird eine Zustandsrelation *R(t)* wie folgt bestimmt:

$$R : T \rightarrow \wp(FS_{State} \times FS_{State})$$

$$R(t) =_{def} \{(\sigma, \sigma') \in FS_{State} \times FS_{State} | \forall e \in E. \sigma(e) \neq (t(\sigma))(e) \Rightarrow \sigma'(e) = (t(\sigma))(e)\}$$

Für jedes Tupel $(\sigma, \sigma') \in R(t)$ gilt, dass es die durch die Anwendung von t auf σ beschriebenen Zustandsänderungen widerspiegelt.

□

Die Markierungsmengen *T* und *Z* leiten wir aus der Zustandsalgebra *FS* ab. *T* ist dabei die Menge, die alle Zustandsabbildungen umfasst, welche der Anwendung (das heißt formale Parameter werden durch konkrete, sortenkonforme Trägermengenelemente substituiert) einer Funktion aus *FS* entspricht. *Z* ist die Menge der Zustandsprädikate, die sich auf analoge Weise aus der Zustandsalgebra *FS* ableiten lässt.

Diese Festlegung von *T* und *Z* ist durch folgende Vorstellung motiviert: Mit den Funktionen in *FS* haben wir alle zulässigen Operationen auf Entitäten festgelegt, und nachdem wir durch Prozesse konkrete Systemabläufe, und nicht Schemata von Abläufen, beschreiben, steht ein (zustandsbezogenes) Prozessereignis für die *Anwendung* einer Operation (und nicht für die Operation).

Die oben definierten Relationen R(t), für $t \in T$, verwenden wir in Definition 2.28, wo wir Prozessereignissen deren Realisierungen in der komponentenbasierten Perspektive zuordnen. Mit R(t) beschreiben wir die Zustandsübergänge, die sich durch Anwendung einer Funktion t ergeben, wobei ein Zustandsübergang weitere, orthogonale und nicht durch t beschriebene Zustandsänderungen umfassen kann. Um diesen Freiheitsgrad zuzulassen, betrachten wir nicht die *Funktion* t, sondern die *Relation* R(t).

Aufbauend auf den Ereignismarkierungen modellieren wir das Verhalten eines Systems (in der ganzheitlichen Perspektive) durch eine Menge von Isomorphieklassen endlich fundierter Prozesse. Wir verwenden hier den Prozessbegriff aus Definition 8.9, der dem Konzept der *labelled partial orders* [Pra86] ähnlich ist. Ebenso in Analogie zu [Pra86] interessieren wir uns nicht für Ereignisse und deren Bezeichnung, sondern für das Auftreten bestimmter *Arten* von Ereignissen, modelliert durch die Ereignismarkierungen, und deren kausale Abhängigkeiten. Daher abstrahieren wir auf Modellebene von Ereignisbezeichnern und betrachten nicht Prozesse, sondern deren Isomorphieklassen.

Mit der oben eingeführte Menge *M* der Markierungen von Prozessereignissen, steht ein Ereignis für das Auftreten bestimmter Zustände, Zustandsübergänge oder Aktionen.

Definition 2.12 (Prozessmenge)

Sei EV eine Menge von Ereignisbezeichnern. Die Menge der Systemprozesse bezeichnen wir mit P, wobei gelte

$P \subseteq [pcs_{fin}(EV,M)]$[15].

□

2.3.3 Reaktionsabbildung

Im Rahmen einer Systemspezifikation interessieren wir uns für bestimmte Ereignisse und welche Reaktionen diese Ereignisse auslösen. Ein Beispiel hierfür sind Ereignisse, die Auslöser von Geschäftsprozessen sind, wie etwa ein Vormerkungswunsch in einem Bibliothekssystem. In Kapitel 3 zeigen wir, dass eine Geschäftsprozessspezifikation im Wesentlichen die Festlegung der zu einem Auslöser gehörenden Reaktionen darstellt. Den Zusammenhang zwischen einem Auslöser und den zugehörigen Reaktionen darauf, erfassen wir in unserem Systemmodell in Form der Abbildung

---

[15] $[pcs_{fin}(EV,M)]$ bezeichnet die Menge der Isomorphieklassen der endlich fundierten Prozesse über EV und M (vergleiche Definition 8.9).

$\text{reak}: \text{M} \rightarrow \wp([\text{pcs}_{\text{fin}}(\text{EV}, \text{M})])$.

Mit Hilfe dieser Abbildung legen wir (indirekt) Reaktionseigenschaften der Systemprozesse auf folgende Weise fest:

Gilt *reak(m) = Q*, dann interpretieren wir dies so, dass ein mit *m* markiertes Ereignis Auslöser für eine der in *Q* beschriebenen alternativen Reaktionen ist (siehe Definition 2.14). Beispielsweise könnte gelten

reak(vormerkungswunsch(Medium m, Leser l)) = {Zuteilung(m, l), Vormerkungsfehler},

wobei *Zuteilung(m, l)* für den Prozess steht, mit dem einem Leser *l* das Medium *m* zugeteilt wird, und *Vormerkungsfehler* für den Prozess steht, der abläuft, wenn ein Fehler bei der Vormerkungsbearbeitung auftritt (vergleiche Beispiel 4.1 bis Beispiel 4.3).

Definition 2.13 (Reaktionsabbildung)

Jedes, mit einer bestimmten Markierung markierte Ereignis, kann bestimmte Reaktionen, das heißt Teilprozesse, auslösen. Diesen Zusammenhang erfassen wir durch die Abbildung

$\text{reak}: \text{M} \rightarrow \wp([\text{pcs}_{\text{fin}}(\text{EV}, \text{M})])$

In Definition 2.14 leiten aus *reak* Lebendigkeitseigenschaften der Prozessmenge P ab.

□

Um in der folgenden Definition die Interpretation der Abbildung *reak* als Anforderung an die Prozessmenge eines Systems zu formulieren, führen wir die beiden Hilfsfunktionen *sometimes* und *leadsto* ein. Beide Funktion sind ähnlich zu den aus der temporalen Logik bekannten, gleichnamigen Operatoren (vergleiche zum Beispiel [MP92]). Informell können wir die beiden Funktionen wie folgt beschreiben:

Für eine Isomorphieklasse *X* von Prozessen und einen Prozess *p* ist *sometimes(X,p)* genau dann erfüllt, falls nach endlich vielen Ereignissen ein Teilprozess *q´* in *p* „vorkommt“, der in *X* liegt. *X* können wir als ein Schema für Prozesse verstehen, da X eine Isomorphieklasse von Prozessen ist und somit von Ereignisbezeichnern abstrahiert wird. Im Kontext der *leadsto* Funktion beschreibt dieses Schema eine korrekte Reaktion auf ein auslösendes Ereignis:

Sei *m* eine Ereignismarkierung, *X* eine Menge von Isomorphieklassen von Prozessen und *p* ein Prozess. Dann ist *leadsto(m, X, p)* genau dann erfüllt, wenn in *p* auf jedes mit *m* markierte Ereignis (nach endlich vielen Ereignissen) ein Teilprozess folgt, der in *X* liegt.

Für die Funktion

$\text{sometimes}: [\text{pcs}_{\text{fin}}(\text{EV}, \text{M})] \times \text{pcs}_{\text{fin}}(\text{EV}, \text{M}) \rightarrow \mathbb{B}$

gelte für alle $\text{X} \in [\text{pcs}_{\text{fin}}(\text{EV}, \text{M})],\ \text{p} \in \text{pcs}_{\text{fin}}(\text{EV}, \text{M})$:

$$\begin{aligned}&\text{sometimes}(X,p) \Leftrightarrow_{def}\\&\exists q.q \sqsubseteq p \wedge\\&\quad(\exists q'.q' \sqsubseteq (p \setminus q) \wedge q' \in X)\end{aligned}$$

Für die Funktion

$$\text{leadsto}: M \times \wp([\text{pcs}_{fin}(EV,M)]) \times \text{pcs}_{fin}(EV,M) \rightarrow \mathbb{B}$$

gelte für alle

$$m \in M,\ X \in \wp([\text{pcs}_{fin}(EV,M)]),\ p \in \text{pcs}_{fin}(EV,M) \text{ mit } p = \left(E_p, \leq_p, \alpha_p\right):$$

$$\begin{aligned}&\text{leadsto}(m,X,p) \Leftrightarrow_{def}\\&\forall q,e.\\&\left(q \sqsubseteq p \wedge e \in \min(p \setminus q) \wedge \alpha_p(e) = m\right) \Rightarrow\\&\exists x \in X.\ \text{sometimes}(x, p \setminus (\Pi_1(q) \cup \{e\}))\end{aligned}$$

Definition 2.14 (Reaktionseigenschaften der Prozessmenge P)

Durch die Abbildung *reak* (aus Definition 2.13) legen wir Reaktionseigenschaften der Systemprozesse P (aus Definition 2.12) fest. Es gelte:

$$\begin{aligned}&\forall m \in M,\ Q \in \wp([\text{pcs}_{fin}(EV,M)]), p \in P.\\&(\text{reak}(m) = Q) \Rightarrow \text{leadsto}(m,Q,p)\end{aligned}$$

□

Wir benötigen die Reaktionsabbildung, um in Abschnitt 4.1 die verschiedenen Formen von Reaktionsspezifikationen definieren zu können.

2.4 Eigenschaften der ganzheitlichen Perspektive

Vor der Diskussion einiger methodisch bedeutsamer Eigenschaften der ganzheitlichen Perspektive, halten wir zusammenfassend fest, dass wir mittels der zustandsbezogenen Ereignismarkierungen, das heißt mittels der Zustandsabbildungen und der Zustandsprädikate, die Integration von fachlichem Zustands- und Verhaltensmodell zu einem umfassenden Modell einer fachlichen Aufgabenstellung schaffen. Hierbei zeigt sich exemplarisch der Grundsatz dieser Arbeit, ausgehend von praxisrelevanten aber informellen Ansätzen, durch Integration formaler aber nicht umfassender Modelle der Informatik, zu einem praxisrelevanten, formalen und umfassenden Ansatz zu kommen. In der ganzheitlichen Perspektive verbinden wir Datenmodellierung mittels Σ-Rechenstrukturen mit Zustandsmodellierung mittels Belegungsabbildungen und Verhaltensmodellierung mittels Partialordnungsstrukturen.

2.4.1 Identifikatorsorten zur Abbildung von Entitätsrelationen

Gängige Spezifikationstechniken fachlicher Daten unterstützen die Definition von Relationen zwischen Entitäten. Notationen wie die Klassendiagramme der UML [OMG01] und Entity-Relationship-Diagramme [Che76] machen dies deutlich.

Dementsprechend muss unser Systemmodell die Möglichkeit bieten, Relationen zwischen Entitäten abzubilden. Wir erreichen dies, indem wir zum einen fordern, dass Σ_{fe} eine Entitätsrechenstruktur (bezüglich den Wertsorten V) ist, so dass eine durch V indizierte Menge von Identifikatorsorten im Modell zur Verfügung steht. Zum anderen definieren wir die Menge der Entitäten E eines Systems als Teilmenge von A_{ID} (der Vereinigung aller Identifikator-Trägermengen, siehe Definition 2.6). Dadurch entsprechen Elemente aus A_{ID} „Referenzen" auf Entitäten. Diese Referenzen können wir (als Parameter von Selektorfunktionen) nutzen, um die verschiedenen notationellen Formen von Assoziationen und Relationen abzubilden.

In Abbildung 2.9 geben wir mögliche Interpretationen einer ungerichteten und einer gerichteten Assoziation an. Die verwendete Notation entspricht den Klassendiagrammen der UML [OMG01] beziehungsweise der algebraischen Spezifikationssprache SPECTRUM [BFG+93].

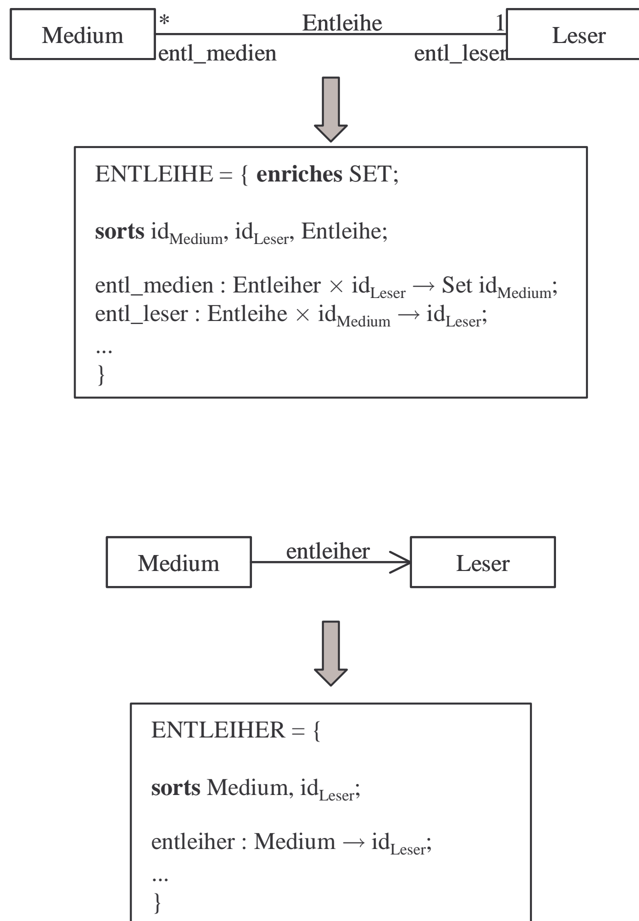


Abbildung 2.9:Mögliche Interpretationen einer ungerichteten und einer gerichteten Assoziation

Für eine detaillierte Diskussion der verschiedenen Arten von Assoziation und der Aggregation im Rahmen eines objektorientierten Ansatzes, verweisen wir auf [Bre98]. Die semantische Fundierung von Entity-Relationship-Diagrammen mittels der Sprache SPECTRUM [BFG+93] ist in [Het95] gegeben.

2.4.2 Entitäten: Kapselung und Verbergen von Information

Hinter der Ableitung der Operationen auf Systemzuständen aus den Operationen der Entitätsalgebra A, wie wir sie mit Definition 2.8 und Definition 2.9 festgelegt haben, stehen folgende Ziele:

- Durch die *Klassifikation* von Entitäten wollen wir festlegen können, welche (atomaren) Zustandsübergänge einer Entität zulässig sind. Dies ist ähnlich zur Datenkapselung in objektorientierten Programmiersprachen, wo die Belegung privater Attribute eines Objektes, das heißt der Objektzustand, nur durch Anwendung von Objektmethoden der Klasse des Objektes verändert werden kann.

Beispielsweise soll der Status eines Mediums nur im Rahmen von entleihe- und rückgabe-Operationen verändert werden, so dass nie der unzulässige Zustand erreicht werden kann, in dem

Status = entliehen

gilt, aber kein Entleiher eingetragen ist.

- Weiterhin wollen wir durch die Klassifikation von Entitäten festlegen können, welche Entitätszustände beobachtbar/unterscheidbar sind (Verbergen/Information-Hiding).

  Beispielsweise soll/muss die Entleih-Historie eines Mediums nicht beobachtbar sein, etwa aus Datenschutzgründen oder um eine effiziente Implementierung zuzulassen.

Wir erreichen die genannten Ziele, indem wir aus den Operationen der Entitätsalgebra *A* entsprechende Zustandsoperationen ableiten. Diese abgeleiteten Operationen definieren wir als die Operationen der Algebra FS eines Entwicklungssystems. Systems. Nur diese lassen wir als Ereignismarkierungen und somit zur Verhaltensspezifikation zu (vergleiche Definition 2.11).

2.4.3 Synchronisation und Lokalität von Ereignismarkierungen

Zustandsbezogene Ereignismarkierungen, das heißt Markierungen aus T oder Z, entsprechen der Anwendung von Funktionen aus FS. Die Funktionen aus FS haben wir so definiert, dass sie *entitätslokal* sind, wobei wir unter Entitätslokalität folgendes verstehen:

> *Der Wert einer Funktion der Form*
>
> $State \rightarrow s$
>
> *ist höchstens von dem „aktuellen" Zustand* einer *Entität abhängig. Im Falle von Funktionen der Form*
>
> $State \rightarrow State$
>
> *bedeutet Entitätslokalität zusätzlich, dass sich der Ergebniszustand vom aktuellen Argument(zustand) nur bezüglich dieser einen Entität unterscheidet.*

Da alle Funktionen aus FS entitätslokal sind, können wir in der ganzheitlichen Perspektive nur entitätslokale „Aussagen" über Systemzustände und Zustandsänderungen machen. Diese Einschränkung halten wir aus methodischer Sicht für sinnvoll, denn dadurch verhindern wir die implizite Spezifikation von Synchronisationsanforderungen. Siehe hierzu folgendes Beispiel:

Beispiel 2.3 (Entitätslokalität von Ereignismarkierungen)

Abbildung 2.10 enthält zwei Prozessbeschreibungen in Form von Aktivitätsdiagrammen der UML. Für die in Abbildung 2.10 verwendeten Bezeichner *l* und *m* gelte:

$m \in id_{Medium}$ und $l \in id_{Leser}$

und für die Funktionen *entleiher*, *is_entleiher*, *rückgabe* und *austragen* gelte

$\text{entleiher} : \text{Medium} \rightarrow \text{id}_{\text{Leser}}$

$\text{is_entleiher} : \text{Medium} \times \text{id}_{\text{Leser}} \rightarrow \text{Bool}$

$\text{rückgabe} : \text{Medium} \rightarrow \text{Medium}$

$\text{austragen}: \text{Leser} \times \text{id}_{\text{Medium}} \rightarrow \text{Leser}$

wobei *entleiher* den aktuellen Entleiher eines Mediums liefert, *is_entleiher* den aktuellen Entleiher mit dem Parameterwert vergleicht, *rückgabe* die durch eine Rückgabe implizierten Wertänderungen eines Mediums beschreibt, und *austragen* das im Parameter angegebene Medium aus der Menge der von einem Leser entliehenen Medien entfernt.

In der linken Prozessbeschreibung wenden wir nur Funktionen aus FS an, die damit entitätslokal sind. Dagegen ist die, in der rechten Beschreibung verwendete, Markierung

(m.entleiher()).austragen(m)

eine komponierte Funktion. Sie liegt nicht in T, da es keine entsprechende Funktion in FS gibt, und stellt daher keine zulässige Markierung dar. Problematisch an dieser Markierung ist, dass nicht nur auf den Zustand einer Entität Bezug genommen wird, sondern auf die Zustände der beiden Entitäten *m* und *m.entleiher()*. Damit werden durch *(m.entleiher()).austragen(m)* die Zustände der beiden Entitäten *implizit* korrelliert und somit „versteckte" Synchronisationsanforderungen formuliert. Dies wird deutlich, wenn wir beispielsweise die Zustandsfolge

$\sigma, \sigma', \sigma''$

mit

$\text{entleiher}(\sigma(m)) = l, \sigma'(m) = \text{rückgabe}(\sigma(m)) \text{ und } \sigma''(l) = \text{austragen}(\sigma'(l),m)$

betrachten. Diese Folge stellt keine „Implementierung" des mit *m.entleiher().austragen(m)* markierten Ereignisses dar, denn im Zustand σ' gilt nicht (mehr): $\text{entleiher}(\sigma'(m)) = l$, da diese Eigenschaft durch die Anwendung von rückgabe() geändert wurde. Damit ist weder (σ,σ'), noch (σ',σ'') in *R((m.entleiher()).austragen(m))*, was gemäss Definition 2.28 von der Interaktionszuordnung(sabbildung) *hi* aber gefordert wäre.

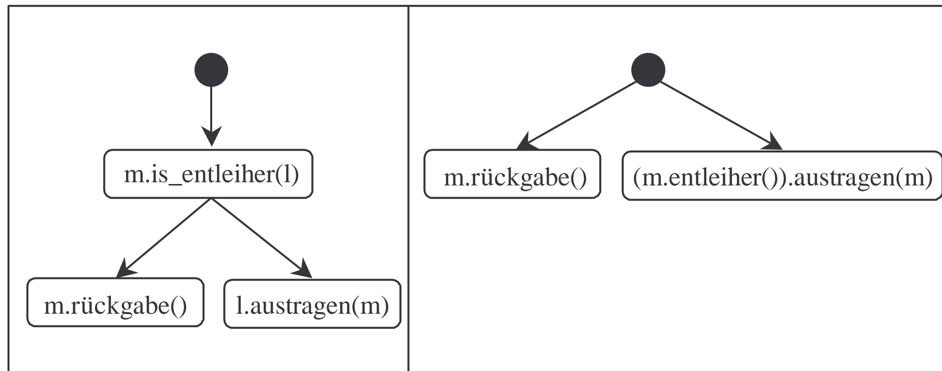


Abbildung 2.10: Prozess-Spezifikationen mit entitätslokaler (linker Fall) beziehungsweise entitätsübergreifender (rechter Fall) Markierung.

□

Die *implizite* Korrelation von Entitätszustände würde zu schwer durchschaubaren Spezifikationen führen, was insbesondere dann problematisch ist, wenn wir korrelierte Entitäten auf unterschiedliche, nebenläufig agierende Komponenten verteilen. Die Beschränkung auf entitätslokale Prozessereignisse halten wir daher aus methodischer Sicht für eine wichtige Entwurfsentscheidung unseres Systemmodells.

2.5 Komponentenbasierte Perspektive

2.5.1 Überblick

Im Rahmen einer Softwareentwicklung ist neben der fachlichen Aufgabenstellung ihre Umsetzung mit Hilfe des zu entwickelnden Softwaresystems zu beschreiben. Mit der ganzheitlichen Perspektive haben wir die Elemente unseres Systemmodells eingeführt, durch die wir eine fachliche Aufgabenstellung beschreiben können. Für die Beschreibung der softwaregestützten Realisierung der Aufgabenstellung führen wir die komponentenbasierte Perspektive des Systemmodells ein.

Während in der ganzheitlichen Perspektive die Strukturierung des Verhaltens und des Zustandsraums des Anwendungssystems im Vordergrund steht, wird das Anwendungssystem in der komponentenbasierten Perspektive in autonom agierende und miteinander interagierende Einheiten strukturiert. Diese Einheiten nennen wir Komponenten.

Eine in Komponenten strukturierte Beschreibung des Anwendungssystems ist notwendig, um die Realisierung einer fachlichen Aufgabenstellung mit Hilfe eines Softwaresystems erfassen zu können. Dies hängt damit zusammen, dass im allge-

meinen das (zu entwickelnde) Softwaresystem die fachliche Aufgabenstellung nicht vollständig, sondern nur einen Teil davon realisiert. Das bedeutet eine Aufgabenteilung zwischen dem Softwaresystem und seiner Umgebung. Wir gehen dabei davon aus, dass Softwaresystem und Umgebung autonom voneinander agieren und an bestimmten „Punkten“ interagieren um Aufgaben zu lösen. Beispielsweise trifft ein Leser unabhängig vom Bibliothekssystem die Entscheidung, ein Buch vormerken zu wollen (Autonomie). Um die Vormerkung auszuführen, meldet sich der Leser am Bibliothekssystem an, das ihn durch einen entsprechenden Dialog führt (Interaktion).

Um die Aufgabenteilung zwischen Softwaresystem und Umgebung beschreiben zu können, müssen wir zwischen dem zu entwickelnden Softwaresystem und seiner Umgebung unterscheiden können. In der komponentenbasierten Perspektive ermöglichen wir dies, indem wir das Softwaresystem und dessen Umgebung durch interagierende Komponenten modellieren. Das Anwendungssystem besteht daher in der komponentenbasierten Perspektive aus mindestens zwei Komponenten, der „Softwaresystemkomponente“ und der „Umgebungskomponente“.

Im allgemeinen werden wir jedoch sowohl das Softwaresystem als auch dessen Umgebung nicht durch jeweils eine einzelne Komponente beschreiben. Vielmehr werden wir diese auf strukturierte Weise durch ein Netzwerk von Komponenten beschreiben, so dass wir nicht von Software- beziehungsweise Umgebungskomponente sprechen, sondern von *Entwicklungsnetzwerk* (für das zu entwickelnde Softwaresystem) und *Umgebungsnetzwerk*. Das durch Entwicklungs- und Umgebungsnetzwerk gebildete Netzwerk beschreibt die (komponentenbasierte) Realisierung des Anwendungssystems. Wir nennen es daher *Realisierungsnetzwerk*.

Abbildung 2.11 skizziert die eingeführten Netzwerke der komponentenbasierten Perspektive und ihre Realisierungsbeziehung zur ganzheitlichen Perspektive.

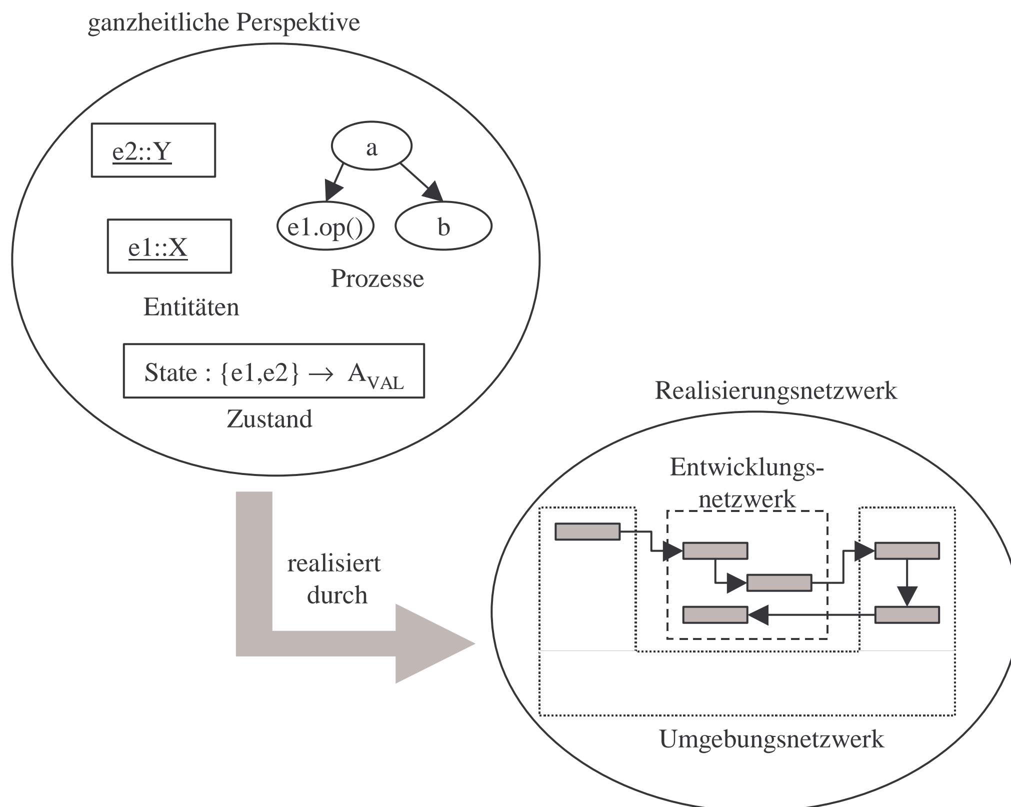


Abbildung 2.11: Ganzheitliche Perspektive und Realisierungsnetzwerk

Die in Komponenten strukturierte Beschreibung des (zu entwickelnden) Softwaresystems und dessen Umgebung kann aus weiteren Gründen sinnvoll und notwendig sein:

- inhärente Verteilung, zum Beispiel bei Internetzugang zum Softwaresystem einer Bibliothek bedeutet räumliche Verteilung von Benutzerschnittstelle und Datenbankserver.
- Anwendung des Entwicklungsprinzips des „Teilen und Herrschens“ (Divide et Impera), um eine komplexe Aufgabenstellung durch Dekomposition in weniger komplexe Bestandteile zu beherrschen.
- modulare Strukturen, um Qualitätskriterien, wie Erweiterbarkeit und Änderbarkeit, zu erfüllen.

Das Realisierungsnetzwerk verstehen wir als das Komponentennetzwerk, durch welches das betrachtete Anwendungssystem realisiert wird. Komponenten können dabei für verschiedenste Aktoren des Anwendungssystems stehen, zum Beispiel für Software- oder Hardware-Komponenten, ebenso wie für Anwenderrollen, zum Beispiel die Rolle des Lesers einer Bibliothek.

Im Rahmen einer Entwicklung beschreiben wir das Realisierungsnetzwerk meist durch mehrere, unterschiedliche Sichten. Ein typisches Beispiel ist die Beschreibung auf verschiedenen Detaillierungsebenen, wie sie sich durch eine Entwicklung nach dem Prinzip der schrittweisen Verfeinerung ergeben (siehe etwa [Wir71]). In

dem deutschen Standardvorgehensmodell, dem V-Modell [BDI97], wird eine hierarchisch strukturierte Beschreibung sogar als bindend vorgegeben, in Form der sogenannten *Erzeugnisstruktur* und den mit ihr assoziierten Entwicklungsprodukten.

Diese Sichten auf das Realisierungsnetzwerk modellieren wir im Systemmodell durch eine endliche Menge von Komponentennetzwerken, die wir *Spezifikationsnetzwerke* nennen, da wir sie im Sinne von Anforderungsspezifikationen an das Realisierungsnetzwerk interpretieren. Zum Beispiel muss das Realisierungsnetzwerk eines Systems dieselbe syntaktische Schnittstelle wie jedes Spezifikationsnetzwerk aufweisen, und zudem eine verfeinertes Netzwerkverhalten bezüglich dieser Schnittstelle.

Mit den Spezifikationsnetzwerken machen wir im Systemmodell die unterschiedlichen Sichten auf das Realisierungsnetzwerk *explizit*, die in einer Entwicklung entstehen. Dadurch schaffen wir (zusammen mit den eindeutig benannten Komponenten der Netzwerke) Bezugspunkte für Produkte, durch die wir unterschiedliche Aspekte *ein und desselben* Systemausschnitts beschreiben. Beispielsweise können wir so in einem Produkt lediglich die syntaktische Schnittstelle der Datenbankschicht eines Systems (repräsentiert durch eine Komponente k eines Spezifikationsnetzwerkes) definieren, während wir in einem anderen Produkt eine Automatensicht der Datenbankschicht, das heißt der Komponente k, formulieren.

Es stellt sich die Frage, auf welche Weise wir Komponenten und Netzwerke modellieren ? Da unsere Arbeit auf die Entwicklung von Software ausgerichtet ist, wählen wir das Konzept einer Komponente so, dass es für die Modellierung von Softwaresystemen geeignet ist. Wir stützen uns dabei auf dem, in der Methode FOCUS [BS00] gegebenen Modell für Komponenten und Netzwerke ab. In Abschnitt 8.4 geben wir die mathematischen Definitionen unseres Komponenten- und Netzwerkbegriffs an. Hier skizzieren wir nur die wichtigsten Eigenschaften informell:

In der komponentenbasierten Perspektive modellieren wir das Anwendungssystem als ein Netzwerk von Komponenten, wobei Komponenten über Kanäle miteinander verbunden werden. Interaktion zwischen den Komponenten findet durch Nachrichtenaustausch über Kanäle statt. Für jeden Kanal legen wir die Menge der Nachrichten fest, die zulässigerweise übertragen werden dürfen und gehen davon aus, dass nur diese Art von Nachrichten übertragen werden. Auf jedem Kanal können Nachrichten nur unidirektional und rein sequentiell überragen werden. Zudem verbindet ein Kanal immer genau eine sendende mit einer empfangenden Komponente (Punkt-zu-Punkt-Verbindung). Broadcasting, beispielsweise, muss mittels entsprechender Broadcast-Komponenten modelliert werden. Nachrichtenaustausch erfolgt auf nachrichten-asynchrone Weise, das heißt der Sender kann Nachrichten immer ungehindert senden und nicht durch den Empfänger blockiert werden. Dies entspricht der Annahme unbeschränkter Pufferkapazitäten.

Eine Komponente ist zum einen durch die Menge seiner Ein- und Ausgabekanäle, die wir unter dem Begriff der syntaktischen Schnittstelle zusammenfassen, charakterisiert. Da wir von einer statischen Systemstruktur ausgehen, bleibt die syntaktische Schnittstelle einer Komponente über dessen Lebensdauer invariant. Das zweite Charakteristikum einer Komponente ist dessen Verhalten. Das beobachtbare Verhalten sind die Ausgaben, die eine Komponente in Abhängigkeit von erhalte-

nen Eingaben macht. Wir modellieren dieses Interaktionsverhalten in Form einer *Verhaltensfunktion*, die Eingabehistorien auf (Mengen von) Ausgabehistorien abbildet. Weiterhin gehen wir davon aus, dass eine Komponente keinen Einfluss darauf hat, welche Eingaben sie erhält (mit der Ausnahme rückgekoppelter Kanäle).

Für eine detaillierte Diskussion verweisen wir auf [BS00]. Die mathematische Definition der im Folgenden verwendeten Komponenten- und Netzwerkbegriffe geben wir in Abschnitt 8.4 an.

Wie in [Sal02] gezeigt, stellt das gewählte Komponenten- und Netzwerkmodell eine geeignete Abstraktion für verbreitete, komponentenbasierte Implementierungsplattformen, wie COM [Box98] und CORBA [Gro98], dar. Zum Beispiel können wir Kanäle als Abstraktion von Referenzen auf/zwischen COM-Komponenten verstehen. Auch für die Beschreibung objektorientierter Systeme ist das Modell geeignet, wie etwa [Rum96] zeigt. Zudem stellt es ein bewährtes, mathematisches Modell für verteilte Systeme dar und mit der Methode FOCUS [BS00] steht ein breites Spektrum von Spezifikationstechniken zur Verfügung, wie zum Beispiel Automaten- und Assumption/Commitment-Spezifikationen.

Neben dem beobachtbaren Komponentenverhalten, das heißt dem reinen Ein-/Ausgabeverhalten einer Komponente, geben wir im Systemmodell für jede Komponente eines Spezifikationsnetzwerkes zusätzlich eine zustandsbezogene Verhaltenssicht an. Hierzu ordnen wir jeder Komponente einen Zustandsraum zu. In Abhängigkeit von Eingaben, macht eine Komponente (in dieser Verhaltenssicht) nicht nur Ausgaben, sondern auch Zustandsübergänge. Um das zustandsbezogene Verhalten auf anschauliche, konstruktive und implementierungsnahe Weise angeben zu können, enthält das Systemmodell für jede Komponente (eines Spezifikationsnetzwerkes) eine Automatensicht.

Die zustandsbezogene Verhaltenssicht ist für unseren Ansatz von besonderer Bedeutung, um einen Zusammenhang zwischen fachlicher Aufgabenstellung und dessen komponentenbasierter Realisierung herstellen zu können. Dies hängt damit zusammen, dass Zustandsänderungen und Zustandsbeobachtungen zentrale Elemente fachlicher Abläufe sind. Wir gehen darauf in Abschnitt 2.5.3 detailliert ein.

Wie erwähnt, formulieren wir durch Spezifikationsnetzwerke Anforderungen an das Realisierungsnetzwerk eines Systems. Neben Anforderungen an die Schnittstelle und das Verhalten, sind wir auch daran interessiert, Anforderungen an die Struktur/Topologie des Realisierungsnetzwerkes formulieren zu können. Beispielsweise unterscheidet [HNS99] zwischen konzeptueller Architektur und technischer Architektur. Während in ersterer Komponenten nur zur strukturierten Beschreibung von Schnittstellenverhalten dienen, fordert letztere Architekturart eine zum Spezifikationsnetzwerk isomorphe Netzwerkstruktur/Topologie der Realisierung.

Im Systemmodell klassifizieren wir Komponenten der Spezifikationsnetzwerke entsprechend den strukturellen Anforderungen, die wir mit einer Komponenten ausdrücken wollen. Mit dem Konzept der sogenannten *Topologiekomponenten* erweitern wir das grundlegende Komponentenkonzept, bestehend aus syntaktischer Schnittstelle und Verhaltensfunktion, um Zustandsraum und topologische Klassifikation. Topologiekomponenten sind somit die geeigneten Bausteine für Spezifikationsnetzwerke.

Zusätzlich zu den aus Topologiekomponenten aufgebauten Spezifikationsnetzwerken, umfasst das Systemmodell Elemente, mit Hilfe derer wir den (globalen) Zustandsraum eines jeden Spezifikationsnetzwerkes (modelliert als eine Algebra, die eine Trägermenge der Netzwerkzustände enthält) sowie eine globale, zustandsbezogene Verhaltenssicht eines Spezifikationsnetzwerkes erfassen. Wir benötigen diese Elemente zur Beschreibung davon, auf welche Weise ein Spezifikationsnetzwerk die, in Form der ganzheitlichen Perspektive gegebene, fachliche Aufgabenstellung einer Entwicklung realisiert. Beispielsweise fordern wir, dass die Zustandsalgebra eines Spezifikationsnetzwerkes homomorph zur fachlichen Zustandsalgebra ist, so dass die Operationen der Algebra des Netzwerkes Realisierungen der Operationen auf dem fachlichen Zustandsraum darstellen.

Die folgende Beschreibung der komponentenbasierten Perspektive gliedert sich wie folgt:

Zunächst führen wir in Abschnitt 2.5.2 das Realisierungsnetzwerk ein, durch das wir die komponentenbasierte Realisierung des Anwendungssystems erfassen. In Abschnitt 2.5.3 definieren wir das Konzept der Topologiekomponente, um diese als Bausteine der in Abschnitt 2.5.4 behandelten Spezifikationsnetzwerke, den verschiedenen Sichten auf das Realisierungsnetzwerk, einzusetzen. Die Automatensicht von Komponenten, zur konstruktiven Verhaltensbeschreibung, ist Gegenstand von Abschnitt 2.5.5. In Abschnitt 2.5.6 gehen wir auf die globalen Zustands- und Verhaltensaspekte von Spezifikationsnetzwerken ein. Analog zur Reaktionsabbildung der ganzheitlichen Perspektive, führen wir in Abschnitt 2.5.7 Reaktionsabbildungen ein, um Reaktionseigenschaften der Spezifikationsnetzwerke zu erfassen.

2.5.2 Realisierungsnetzwerk

Das Realisierungsnetzwerk verstehen wir als das Komponentennetzwerk, durch welches das betrachtete Anwendungssystem realisiert wird. Komponenten können dabei für verschiedenste Aktoren des Anwendungssystems stehen, zum Beispiel für Software- oder Hardware-Komponenten, ebenso wie für Anwenderrollen, zum Beispiel die Rolle des Lesers einer Bibliothek.

Als Grundlage für die verschiedenen Netzwerke des Systemmodells dienen das Komponenten- und Netzwerkmodell aus Definition 8.19 und Definition 8.21. Im Systemmodell führen wir zunächst die Menge N der Nachrichten eines Systems sowie die Abbildung *ctype* ein, durch die wir jedem Kanal die Menge von Nachrichten zuordnen, die auf ihm übertragen werden können.

Definition 2.15 (Nachrichtenuniversum und typisierte Kanäle)

Mit

$$N$$

bezeichnen wir eine abzählbare Menge von Nachrichten. Die Nachrichten, die auf einem Kanal aus CH übertragen werden können, legen wir mittels der Abbildung

$ctype : CH \to \wp(N)$

fest.

□

Definition 2.16 (Realisierungsnetzwerk)

Die Realisierung des Anwendungssystems umfasst die Realisierung des zu entwickelnden Softwaresystems sowie dessen Umgebung. Die Realisierung des Softwaresystems modellieren wir durch das Komponentennetzwerk

$dnw \in NW(CH)$ [16] (dnw steht für development network)

die Realisierung der Umgebung durch

$enw \in NW(CH)$ (enw steht für environment network).

wobei gelte:

$\forall c \in dnw, c' \in enw. \neg(c = c')$ (Disjunktheit der Netzwerke)

Die Realisierung des Anwendungssystems, beschrieben durch das Netzwerk

$nw \in NW(CH)$

ergibt sich aus der Komposition der beiden Netzwerke:

$nw =_{def} dnw \cup enw$

□

2.5.3 Topologiekomponenten und -netzwerke

In diesem Abschnitt führen wir das Konzept der Topologiekomponente und des Topologienetzwerkes ein, die beide auf den in Definition 8.19 beziehungsweise Definition 8.21 gegebenen Komponenten- beziehungsweise Netzwerk-Konzept aufbauen und diese so erweitern, dass mit Topologiekomponenten geeignete Bausteine für die Spezifikationsnetzwerke des Systemmodells zur Verfügung stehen. Wie bereits erwähnt, dienen Spezifikationsnetzwerke dazu, Eigenschaften des Realisierungsnetzwerkes anzugeben. Das heißt, ein Spezifikationsnetzwerk entspricht einer, in Form eines Netzwerkes gegebenen Sicht auf das Realisierungsnetzwerk.

Der Komponentenbegriff aus Definition 8.19 umfasst die syntaktische Schnittstelle einer Komponente sowie das Interaktionsverhalten bezüglich dieser Schnittstelle. Wir verwenden diesen Komponentenbegriff, da damit genau die *beobachtbaren* Komponenteneigenschaften erfasst werden. Die beobachtbaren Eigenschaften sind wesentlich für die Perzeption (etwa aus Anwendersicht) und Komposition von Komponenten.

[16] NW(CH) steht für die Menge der Komponentennetzwerke über der Kanalmenge CH, vergleiche Definition 8.21.

Um geeignete Bausteine für die *Spezifikation* des Realisierungsnetzwerkes zur Verfügung zu haben, erweitern wir den Komponentenbegriff um einen expliziten Zustandsraum, eine zustandsbezogene Verhaltensmodellierung sowie um topologische Bedeutung.

Die Erweiterung um Zustandsaspekte hat folgende Gründe:

- Die Angabe eines expliziten Komponentenzustands erlaubt es, Zustandsaspekte von Interaktionen (wie die, durch Eingaben implizierte Änderung von Daten- und/oder Kontrollzustand, sowie die Zustandsabhängigkeit von Ausgaben) auf anschauliche Weise zu beschreiben. Eine anschauliche Verhaltensspezifikation, die auf einem expliziten Zustandsbegriff basiert, ist insbesondere durch Automaten/Transitionssysteme aufgrund ihres konstruktiven Charakters möglich. Transitionssysteme geben an, wie schrittweise Verhalten „generiert" wird (vergleiche Abschnitt 2.5.5). Der verbreitete Einsatz von Transitionssystemen zur Spezifikation des Verhaltens verteilter Systeme, seien sie komponentenorientiert (zum Beispiel [CD94, Krü01, PTL+99, Rum96]) oder nicht (etwa [Lam94]), weist auf die Bedeutung dieser Beschreibungsform hin.
- Für unseren Ansatz von größter Bedeutung ist die Rolle der zustandsbezogenen Sicht als *Bindeglied zwischen der reinen Interaktionssicht und der ganzheitlichen Perspektive*, deren Prozesse zustandsbezogene Ereignisse (Zustandsbeobachtungen und –übergänge) umfassen. Dies hängt damit zusammen, dass nur durch Angabe eines expliziten Komponentenzustands und dessen Bezug zum Zustand der ganzheitlichen Perspektive, die Bedeutung von Interaktionen bezüglich fachlicher Zustandsaspekte dargestellt werden kann. Wir gehen darauf detailliert in Abschnitt 2.6.2 ein.
- Weiterhin ist die zustandsbezogene Sicht aufgrund ihrer konzeptuellen Nähe zu gängigen Implementierungstechnologien (von Informationssystemen) eine wichtige Entwicklungssicht. Als Beispiel seien hier objektorientierte Entwicklungsumgebungen genannt, in denen durch („private") Attribute ein Zustandsraum aufgespannt wird, der durch Interaktion, das heißt durch Methodenaufruf, modifiziert und „abgefragt" werden kann. Eine zustandsbezogene Komponentensicht kann somit als Vorlage/Feinentwurf für eine zustandsbasierte Implementierung dienen.

Die Erweiterung von Komponenten um topologische Information bedeutet konkret, dass wir aus topologischer Sicht verschiedene Komponentenarten unterscheiden. Abhängig von dieser topologischen Klassifikation, verbinden wir mit einer Komponente unterschiedliche Anforderungen an die Netzwerkstruktur/Topologie des Realisierungsnetzwerkes. Damit haben wir zum Beispiel die Möglichkeit zu unterscheiden, ob eine Komponente eines Spezifikationsnetzwerkes (nur) für einen Teil des Gesamtverhaltens steht, oder aber eine strukturell exakte Abbildung einer Komponente des Realisierungsnetzwerkes ist, wie dies etwa für die Formulierung der Anforderung, eine gegebene Legacy-Komponente in der Realisierung einzubinden, angemessen ist. Detailliert gehen wir in Abschnitt 2.5.4 auf die topologischen Aspekte ein.

Abbildung 2.12 illustriert die unterschiedlichen Rollen von „einfachen" Komponenten und von Topologiekomponenten im Systemmodell. Zu beachten sind die

Unterschiede in der internen Netzwerkstruktur, bei Übereinstimmung der syntaktischen Schnittstelle, der drei dargestellten Netzwerke.

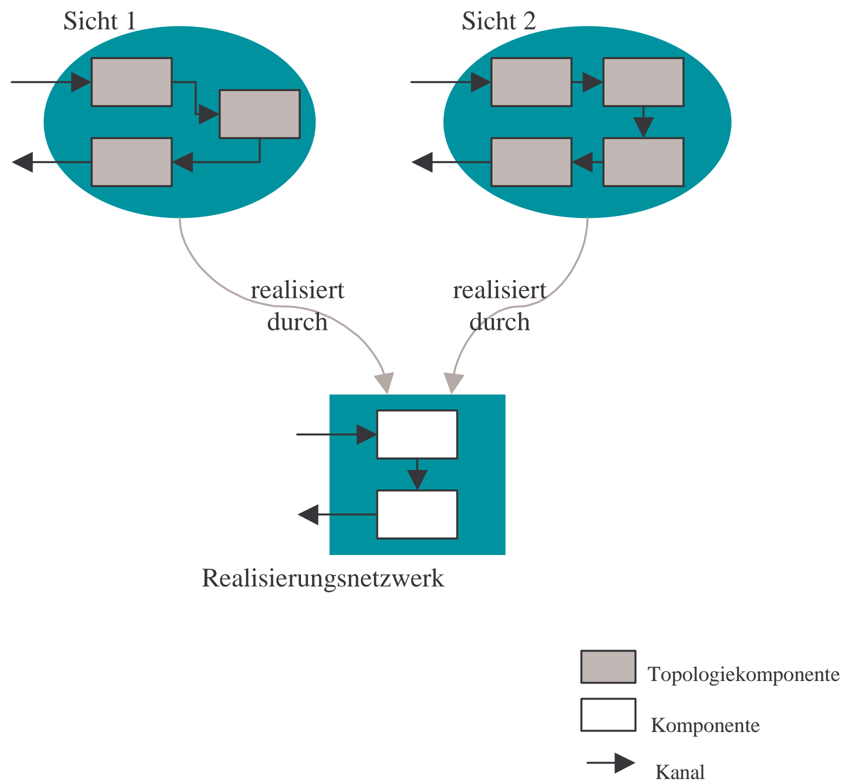


Abbildung 2.12: Rolle von Komponenten und Topologiekomponenten im Systemmodell.

Formal definieren wir den Begriff der Topologiekomponente und wie folgt:

Definition 2.17 (Topologiekomponente)

Sei CH das Kanaluniversum. Eine *Topologiekomponente* ist ein 4-Tupel

((I,O,f), t, S, E)

bestehend aus

- einer Komponente (gemäss Definition 8.19) $(I, O, f) \in KP(CH)$
- einer topologischen Klasse $t \in TClass$, mit

  $TClass =_{def} \{real, interface, behavior\}$
- eine nichtleeren Menge von Zuständen S sowie

- einer Menge $E \subseteq (IS_{I \cup O,S})^{\infty}$ von Ausführungsfolgen[17].

Für f und E gelte folgender Zusammenhang:

$$\forall i \in \vec{I}, o \in \vec{O}.$$
$$o \in f(i) \Leftrightarrow \exists s \in S^{\infty}, \psi \in (IS_{I \cup O,S})^{\infty}.$$
$$\psi \in E \wedge$$
$$\Pi_{Istream}(\psi)|_I = i \wedge$$
$$\Pi_{Istream}(\psi)|_O = o \wedge$$
$$\Pi_{Sstream}(\psi) = s$$

$\Pi_{Istream}$ und $\Pi_{Sstream}$ stehen dabei für die in Definition 8.24 definierten Projektionen auf die in ψ enthaltenen Interaktions- beziehungsweise der Zustandshistorien. Die Menge aller Topologiekomponenten über CH bezeichnen wir mit

TKP(CH).

Analog zu Definition 8.21 definieren wir für jede Topologiekomponente $c \in TKP(CH)$:

$$component(c) =_{def} \Pi_1(c),$$
$$in(c) =_{def} in(\Pi_1(c)),$$
$$out(c) =_{def} out(\Pi_1(c)),$$
$$channels(c) =_{def} in(c) \cup out(c),$$
$$behav(c) =_{def} behav(\Pi_1(c)),$$
$$tclass(c) =_{def} \Pi_2(c),$$
$$states(c) =_{def} \Pi_3(c),$$
$$exec(c) =_{def} \Pi_4(c)$$

□

Abbildung 2.13 veranschaulicht den Aufbau einer Topologiekomponente.

---

[17] für eine Kanalmenge C und eine Zustandsmenge S bezeichnen wir mit $IS_{C,S}$ die Menge der sogenannten Interaktions-Zustands-Tupel gemäss Definition 8.22

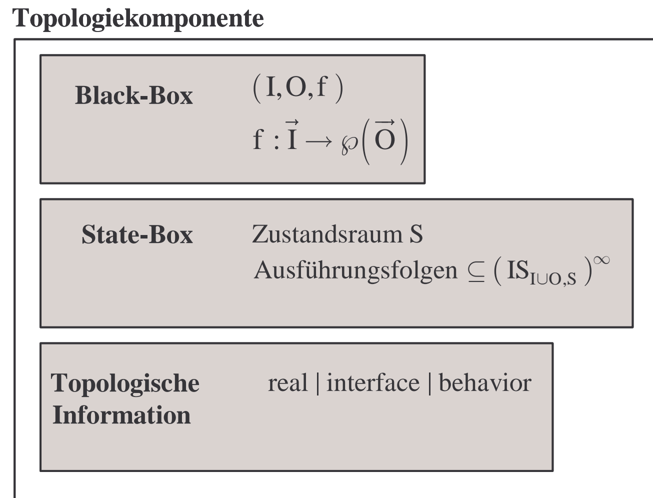


Abbildung 2.13: Elemente einer Topologiekomponente.

Die in einer Topologiekomponente enthaltene „einfache" Komponente gibt die syntaktische Schnittstelle sowie das Interaktionsverhalten an. Damit entspricht sie der Schnittstellen-/Black-Box-Sicht der Topologiekomponente.

Um die zustandsbezogene Verhaltenssicht einer Topologiekomponente in Form ihrer Ausführungsfolgen angeben zu können, führen wir den Zustandsraum der Komponente in Form einer nichtleeren Zustandsmenge ein.

Als Modell der Ausführungsfolgen verwenden wir die in Definition 8.22 eingeführten Interaktions-Zustandsströme (synonym: IS-Ströme), das heißt Ströme deren Elemente 2-Tupel sind, bestehend aus einem Interaktionsmuster (für eine gegebene Kanalmenge CH) und einem Zustand (aus einer gegebenen Zustandsmenge S). Ein (unendlicher) IS-Strom beschreibt die Verhaltenshistorie einer Komponente, wobei wir ein Element $(i,s) \in IS_{CH,S}$ des Stroms als die Interaktion *i* und den Zustand *s* der Komponente zu einem bestimmten Zeitpunkt/Takt verstehen. Damit haben wir Zustand und Interaktion korreliert. Dies erlaubt es uns, kausale Abhängigkeiten zwischen Zustand und Interaktion (und umgekehrt) festzulegen (siehe Beispiel 2.4).

Beispiel 2.4 (Korrelation von Zustand und Interaktion in IS-Strömen)

Eine typische Eigenschaft, in der Zustand und Interaktion korreliert sind, ist die Forderung, dass eine Ausgabe o_z immer nur dann erfolgt, falls das Netzwerk (im Takt) vorher in einem Zustand war, in dem *z(s)* galt, wobei *z* ein Zustandsprädikat sei:

$$\forall n \in \mathbb{N}_+, \varphi \in IS^\infty .$$
$$o_z \in \Pi_1(\varphi @ (n+1)) \Rightarrow z(\Pi_2(\varphi @ (n)))$$

Mit obiger Eigenschaft ist es möglich, Rückschlüsse von der Komponentenausgabe (o_z) auf den (vorangegangenen) Komponentenzustand zu ziehen, so dass beispielsweise die Übertragung einer Nachricht o_z als „Realisierung einer Zustandsbeobachtung", in der ganzheitlichen Perspektive modelliert durch ein mit *z* markiertes Pro-

zessereignis, verstanden werden kann (ein Beispiel für z wäre etwa *m.eqentleiher(l)*).

□

Mit obiger Definition der Topologiekomponenten fordern wir, dass das durch die Verhaltensfunktion beschriebene, reine Interaktionsverhalten und das zustandsbezogene Verhalten einer Topologiekomponente auf folgende Weise übereinzustimmen haben:

- zu jedem Interaktionsverhalten gibt es (mindestens) eine passende Ausführungsfolge ($\Rightarrow$). Im allgemeinen werden mehrere Ausführungsfolgen zu einem Interaktionsverhalten existieren, zum Beispiel, falls in einem die Ausführungsfolgen spezifizierenden Automaten unterschiedliche Transitionspfade zu einer Eingabehistorie existieren.
- jede, in einer Ausführungsfolge enthaltene Kombination von Ein- und Ausgabehistorie ist auch durch die Verhaltensfunktion beschrieben ($\Leftarrow$).

In Abschnitt 8.4.3 ordnen wir Automaten das von ihnen generierte Verhalten in Form von (Mengen von) IS-Strömen zu. Wir betrachten Automaten als ein ideales Modell, um Zusammenhänge zwischen Zustands- und Interaktionsverhalten einer Komponente zu beschreiben, so dass wir im Systemmodell jeder Topologiekomponente einen Automaten zuordnen, welcher die Ausführungsfolgen der Komponente „generiert" (siehe Abschnitt 2.5.5).

Aufbauend auf dem Konzept der Topologiekomponente bilden wir das Konzept des Topologienetzwerkes. Die Spezifikationsnetzwerke des Systemmodells sind Topologienetzwerke.

Definition 2.18 (Topologienetzwerke)

Die Menge der *Topologienetzwerke* definieren wir als die Menge von endlichen Mengen von Topologiekomponenten, die ein Netzwerk mit Punkt-zu-Punkt Verbindungen gemäss Definition 8.21 bilden:

$$\mathrm{TNW}(\mathrm{CH}) =_{\mathrm{def}} \left\{ \mathrm{x} \in \wp_{\mathrm{fin}}(\mathrm{TKP}(\mathrm{CH})) \middle| \mathrm{components}(\mathrm{x}) \in \mathrm{NW}(\mathrm{CH}) \right\}.$$

wobei

$$\mathrm{components}(\mathrm{x}) =_{\mathrm{def}} \{ \mathrm{c} \in \mathrm{KP}(\mathrm{CH}) \mid \exists \mathrm{c}' \in \mathrm{x}.\mathrm{c} = \mathrm{component}(\mathrm{c}') \}$$

sei.

Die syntaktische Schnittstelle und das (Interaktions-)Verhalten bezüglich dieser Schnittstelle ergibt sich für ein Topologienetzwerk *nw* aus dem durch *components(nw)* gebildeten Komponentennetzwerk. Für jedes Topologienetzwerk $\mathrm{nw} \in \mathrm{TNW}(\mathrm{CH})$ definieren wir folgende abkürzende Schreibweisen:

$$\mathrm{in}(\mathrm{nw}) = \mathrm{in}(\mathrm{components}(\mathrm{nw})),$$
$$\mathrm{out}(\mathrm{nw}) = \mathrm{out}(\mathrm{components}(\mathrm{nw})),$$
$$\mathrm{behav}(\mathrm{nw}) = \mathrm{behav}(\mathrm{components}(\mathrm{nw}))$$

Analoges gilt für die Menge der Kanäle *channels(nw)*:

$$\text{channels}(\text{nw}) =_{\text{def}} \text{channels}(\text{components}(\text{nw}))$$

Die Zustandsmenge bezeichnen wir mit *states(nw)*, die wir als die Menge aller Abbildungen definieren, die jeder Komponente einen Wert aus der Zustandsmenge der jeweiligen Komponente zuordnet:

$$\text{states}(\text{nw}) =_{\text{def}} \left\{ s : \text{nw} \to \bigcup_{c \in \text{nw}} \text{states}(c) \,\middle|\, \forall c \in \text{nw}.s(c) \in \text{states}(c) \right\}$$

Die Ausführungsfolgen nennen wir *exec(nw)* und definieren sie als die Menge aller möglichen Kombinationen der Ausführungsfolgen der Komponenten des Netzwerkes:

$$\begin{aligned}\text{exec}(\text{nw}) =_{\text{def}} \Big\{ \varphi \in \left(\text{IS}_{\text{channels}(\text{nw}),\text{states}(\text{nw})}\right)^{\infty} \Big| \\ \forall c \in \text{components(nw)}.\exists \psi \in \text{exec}(c).\forall n \in \mathbb{N}_{+}. \\ \Pi_1(\varphi @ n)|_{(\text{in}(c) \cup \text{out}(c))} = \Pi_1(\psi @ n) \wedge \\ (\Pi_2(\varphi @ n))(c) = \Pi_2(\psi @ n) \}\end{aligned}$$

□

2.5.4 Spezifikationsnetzwerke

Im allgemeinen beschreiben wir ein System im Rahmen einer Entwicklung aus mehreren, unterschiedlichen Sichten, um unterschiedliche, im System manifestierte Aspekte und Strukturen klar herauszustellen. Ein typisches Beispiel ist die Beschreibung eines Systems auf verschiedenen Detaillierungsebenen, wie sie sich durch eine Entwicklung nach dem Prinzip der schrittweisen Verfeinerung ergeben (siehe etwa [Wir71]). In [BDI97] wird eine hierarchisch strukturierte Beschreibung des Systems sogar als bindend vorgegeben, in Form der sogenannten *Erzeugnisstruktur* und den mit ihr assoziierten Entwicklungsprodukten.

Ein weiteres Beispiel stellen die zwei verschiedenen Architektursichten dar, die in [HNS99] vorgeschlagen werden:

- Die konzeptuelle Architektur, anhand derer Eigenschaften des Schnittstellenverhaltens eines Systems auf strukturierte Weise, in Form eines Komponentennetzwerkes, festgelegt werden. Die Dekomposition des Systems in Komponenten folgt hierbei fachlichen Aspekten und hat keinen bindenden Charakter für die Topologie der Realisierung.
- Die Modul-Architektur, die eine Dekomposition in Komponenten, die in diesem Fall Module genannt werden, angibt, welche technischen Kriterien folgt. Die durch die Module vorgegebene Topologie und die vorgegebenen Schnittstellen sind in der Realisierung einzuhalten. Die Modul-Architektur muss nicht notwendigerweise eine Verfeinerung der konzeptuellen Architektur darstellen, etwa wenn dies aus Performanzgründen oder aufgrund bestimmter Qualitätskriterien sinnvoll erscheint.

In unserem Ansatz lassen sich diese Architektursichten durch zwei entsprechende Spezifikationsnetzwerke erfassen. Allgemein bilden wir in unserem Ansatz eine, in Form eines Netzwerkes gegebene, Sicht/Spezifikation des Realisierungsnetzwerkes durch ein Spezifikationsnetzwerk ab. Die beiden genannten Architekturarten aus [HNS99] machen deutlich, warum wir uns nicht auf eine Hierarchie spezifikatorischer Sichten, in der eine Komponente wieder in Form eines Netzwerkes beschrieben werden kann, beschränken, wie dies etwa im V-Modell der Fall ist. Auf diese Weise erhalten wir den Freiraum, beispielsweise sowohl die horizontale Schichtung eines Softwaresystems, bestimmt durch aufeinander aufbauende technische Funktionalitäten, als auch die vertikale Schichtung, bestimmt durch die angebotenen fachlichen Funktionalitäten, jeweils anhand eines Spezifikationsnetzwerkes darzustellen.

Da wir im allgemeinen eine bestimmte Sicht nicht vollständig durch *ein* Produkt beschreiben, sondern durch eine *Menge* von Produkten, wobei jedes dieser Produkte einen Teil(aspekt) der Sicht erfasst, erfassen wir alle Sichten und ihre Elemente *explizit* im Systemmodell. Auf diese Weise können wir zum Beispiel mit einem Produkt nur die statische Struktur einer bestimmten Sicht charakterisieren, in einem anderen Produkt Verhaltenseigenschaften dieser Sicht, das heißt des die Sicht repräsentierenden Spezifikationsnetzwerkes. Außerdem wird es dadurch möglich, ein Produkt zum Beispiel als die „feinste" Architektur auszuzeichnen, indem wir fordern, dass das Produkt jenes Spezifikationsnetzwerk (eines Entwicklungssystems) beschreibt, welches Verfeinerung jedes anderen Spezifikationsnetzwerkes ist (vergleiche Abschnitt 6.4).

Definition 2.19 (Spezifikationsnetzwerke)

Das Systemmodell umfasst eine endliche Anzahl von Spezifikationsnetzwerken. Um ein bestimmtes Spezifikationsnetzwerk einfach referenzieren zu können, modellieren wir die Menge dieser Netzwerke durch eine mit natürlichen Zahlen indizierte Menge. Die Indexmenge legen wir fest, als das geschlossene Intervall

$[1, ..., n_{snw}]$,

wobei

$n_{snw} \in \mathbb{N}_+$

die Anzahl der Spezifikationsnetzwerke eines Systems angibt.

Die Menge der Spezifikationsnetzwerke definieren wir als

$$SNW =_{def} \{snw_i \in TNW(CH) | i \in [1,...,n_{snw}]\}$$

Jedes Spezifikationsnetzwerk ist aus einer endlichen Menge eindeutig bezeichneter Topologiekomponenten aufgebaut. Die eindeutige Bezeichnung einer Komponente erlaubt uns die einfache Referenzierung derselben, um so durch ein Entwicklungsprodukt Eigenschaften einer *bestimmten* Komponente angeben zu können. Dies ist insbesondere dann wichtig, wenn nicht alle Eigenschaften einer Komponente durch *ein* Produkt festgelegt werden, sondern durch eine *Menge* von Produkten. Für jedes Spezifikationsnetzwerk snw_i, $i \in [1,...,n_{snw}]$, steht

$$K_i \subseteq KID, \; |K_i| \in \mathbb{N}_+$$

für die nichtleere, endliche Menge der Komponenten(bezeichner) des Netzwerkes.

Die Menge

$$K =_{def} \bigcup_{i \in [1,...,n_{snw}]} K_i$$

steht für die Gesamtheit aller in Spezifikationsnetzwerken verwendeten Komponentenbezeichner.

Mittels der Abbildung

$$comp : K \rightarrow TKP(CH)$$

ordnen wir jedem Komponentenbezeichner eine Topologiekomponente zu.

Damit definieren wir jedes Spezifikationsnetzwerk $snw_i, i \in [1,...,n_{snw}]$ durch

$$snw_i =_{def} \{c \in TKP(CH) | \exists k \in K_i . c = comp(k)\}.$$

Zu beachten ist, dass die Abbildung *comp* so definiert sein muss, dass (für alle i) die durch snw_i bezeichnete Komponentenmenge nicht eine beliebige Menge, sondern ein Komponentennetzwerk gemäss Definition 8.21 darstellt (und so die Eigenschaft der Punkt-zu-Punkt-Verbindungsstruktur eingehalten wird).

Im Folgenden bezeichnen wir für alle $k \in K$ mit

$$((I_k, O_k, behavior_k), tclass_k, states_k, exec_k) \in TKP(CH)$$

die durch *comp(k)* dem Bezeichner *k* zugeordnete Komponente.

Weiterhin verwenden wir für alle $snw_i, i \in [1,...,n_{snw}]$ folgende abkürzende Schreibweise

$$IS_i =_{def} IS_{channels(snw_i),states(snw_i)}$$

□

Anhand eines exemplarischen Spezifikationsnetzwerkes snw_j illustriert Abbildung 2.14 die mit obiger Definition gegebene Bedeutung der Funktion *comp* zur Festlegung von Spezifikationsnetzwerken, sowie die Partitionierung von snw_j in Entwicklungsnetzwerk dnw_j und Umgebungsnetzwerk enw_j mittels der Funktion *develop*, wie wir sie mit der anschließenden Definition einführen.

Wie erwähnt, gehen wir davon aus, dass im Rahmen einer Entwicklung sowohl das zu entwickelnde Softwaresystem als auch dessen Umgebung betrachtet werden. Auf diese Weise können zum Beispiel der Entwicklung zugrunde liegende Annahmen über die Systemumgebung erfasst werden.

Dementsprechend modellieren wir in der komponentenbasierten Perspektive die Realisierung des Anwendungssystems als ein Komponentennetzwerk, dass sich aus zwei (bezüglich ihrer Komponentenmengen disjunkten) Netzwerken zusammensetzt: dem Netzwerk, welches die Realisierung des zu entwickelnden Softwaresystem beschreibt und dem Netzwerk, welches die Umgebung des Softwaresystems darstellt. Entsprechend der Strukturierung des Realisierungsnetzwerkes in Entwicklungsnetzwerk *dnw* und Umgebungsnetzwerk *enw* (siehe Definition 2.16) müssen

wir auch in den Spezifikationsnetzwerken zwischen dem Softwaresystem- und dem Umgebungsanteil unterscheiden:

Definition 2.20 (Entwicklungs- und Umgebungsnetzwerke)

Durch die Abbildung

$$develop : K \to \mathbb{B}$$

zeichnen wir die zu entwickelnden Komponenten aus. Dadurch wird die Bezeichnermenge $K_i, i \in [1,...,n_{snw}]$ in Bezeichner zu entwickelnder Komponenten und in Bezeichner von Komponenten, welche die Umgebung bilden, partitioniert:

$$dK_i =_{def} \{k \in K_i \mid develop(k)\}$$

$$eK_i =_{def} \{k \in K_i \mid \neg develop(k)\}$$

Dadurch wird wiederum jedes Spezifikationsnetzwerk $snw_i, i \in [1,...,n_{snw}]$ in einen zu entwickelnden Softwareanteil dnw_i und einen Umgebungsanteil enw_i partitioniert:

$$dnw_i =_{def} \{c \in TKP(CH) \mid \exists k \in K_i . c = comp(k) \wedge develop(k)\}$$

und

$$enw_i =_{def} \{c \in TKP(CH) \mid \exists k \in K_i . c = comp(k) \wedge \neg develop(k)\}$$

mit

$$dnw_i, enw_i \in TNW(CH)$$

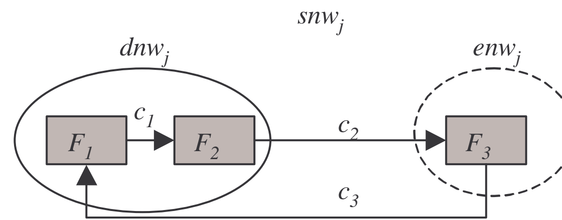
und

$$dnw_i \cap enw_i = \varnothing$$

□

Zu beachten ist, dass wir mit obiger Definition ein Spezifikationsnetzwerk tatsächlich in zwei *disjunkte* Komponentenmengen partitionieren, die, aufgrund der Disjunktheit und der Netzwerkeigenschaft des Spezifikationsnetzwerkes, auch wieder Netzwerke bilden. Diese Eigenschaften sind notwendig, da wir durch *dnw* beziehungsweise *enw* den zu entwickelnden beziehungsweise nicht zu entwickelnden Anteil des Anwendungssystems abbilden und somit die Systemgrenze zwischen dem zu entwickelnden Softwaresystem und seiner Umgebung bestimmen. Eine eindeutige Systemgrenze besteht nur dann, wenn für jede Komponente eindeutig bestimmt ist, ob sie Teil des Softwaresystems oder der Umgebung ist (Disjunktheit von dnw und enw).

Abbildung 2.14 illustriert die Bedeutung der Funktionen *comp* und *develop*.



Systemmodell-Elemente
K_j, comp, develop bzgl. *snw_j*:

K_j= {k_1, k_2, k_3},

comp(k_1) = ({c_3},{c_1},F_1, …),
comp(k_2) = ({c_1},{c_2},F_2, …),
comp(k_3) = ({c_2},{c_3},F_3, …),

develop(k_1)= develop(k_2)= true,
develop(k_3)= false

Abbildung 2.14: Exemplarisches Spezifikationsnetzwerk mit zugehörigen Systemmodellelementen

Wie bereits erwähnt modellieren wir durch die Spezifikationsnetzwerke die unterschiedlichen Sichten auf das Realisierungsnetzwerk eines Systems, wobei jede dieser Sichten bestimmte Eigenschaften des Realisierungsnetzwerkes festlegt. Mit der folgenden Definition legen wir diese Rolle der Spezifikationsnetzwerke im Systemmodell fest.

Definition 2.21 (Spezifikations- und Realisierungsnetzwerk)

Jede Realisierung des zu entwickelnden Softwaresystems und dessen Umgebung muss die, durch die Menge der Spezifikationsnetzwerke formulierten Anforderungen erfüllen. Es muss daher gelten:

$$dnw \in \bigcap_{i \in [1,\ldots,n_{snw}]} nw_real(dnw_i)$$

und

$$enw \in \bigcap_{i \in [1,\ldots,n_{snw}]} nw_real(enw_i)$$

wobei die Abbildung

$$nw_real : TNW(CH) \rightarrow \wp(NW(CH))$$

jedem Topologienetzwerk die Menge von Komponentennetzwerken zuordnet, welche das Topologienetzwerk realisieren. Realisieren bedeutet, dass

- die syntaktische Schnittstelle der Realisierung mit der Spezifikation übereinstimmt/identisch ist,

- das Black-Box-Verhalten der Realisierung das Verhalten der Spezifikation verfeinert/impliziert, und
- die durch die topologische Klassifikation der Topologiekomponenten formulierten topologischen Eigenschaften erfüllt sind.

Für alle $x \in TNW(CH)$ definieren wir die Abbildung *nw_real* wie folgt:

$$nw_real(x) =_{def} \begin{cases} \{\varnothing\}, & \text{falls } x = \varnothing \\ \{y \in NW(CH) | \\ \quad blackbox_consistent(components(x), y) \wedge \\ \quad \forall c \in x.\ topology(component(c), tclass(c))(y)\}, & \text{sonst} \end{cases}$$

mit

$$blackbox_consistent : NW(CH) \times NW(CH) \rightarrow \mathbb{B}$$

wobei

wobei für alle x, x´ $\in$ NW(CH) gelte

$$\begin{aligned} & blackbox_consistent(x, x´) \Leftrightarrow_{def} \\ & in(x´) = in(x) \wedge \\ & out(x´) = out(x) \wedge \\ & \forall i \in \overrightarrow{in(x)}.behav(x´)(i) \subseteq behav(x)(i) \end{aligned}$$

und

$$topology : (KP(CH) \times TClass) \rightarrow (NW(CH) \rightarrow \mathbb{B})$$

wobei für alle (I,O,F) $\in$ KP(CH), nw $\in$ NW(CH) gelte

$$\begin{aligned} topology((I,O,F), real)(nw) \Leftrightarrow_{def} \ & \exists c \in nw. \\ & in(c) = I \wedge \\ & out(c) = O \wedge \\ & bconsistent((I,O,F), \{c\}) \end{aligned}$$

$$\begin{aligned} topology((I,O,F), interface)(nw) \Leftrightarrow_{def} \ & \exists x \subseteq nw. \\ & I \subseteq in(x) \wedge \\ & O \subseteq out(x) \wedge \\ & bconsistent((I,O,F), x) \end{aligned}$$

$$topology(k, behavior)(nw) =_{def} true$$

mit

$$bconsistent : KP(CH) \times NW(CH) \rightarrow \mathbb{B}$$

wobei für alle $c \in KP(CH), w \in NW(CH)$ gelte:

$$\text{bconsistent}(c, w) \Leftrightarrow_{\text{def}}$$
$$\forall \varphi \in \overrightarrow{\text{in}(w)}.(\text{behav}(w)(\varphi))|_{\text{out}(c)} \subseteq \text{behav}(c)\left(\varphi|_{\text{in}(c)}\right)$$

Dabei bezeichnen wir mit $y|_C$ die Einschränkung einer Belegung $y \in \overrightarrow{C'}, C \subseteq C'$, auf die Kanäle in C.

□

Abbildung 2.15 illustriert den mit obiger Definition gegebenen Zusammenhang zwischen Spezifikationsnetzwerken und dem Realisierungsnetzwerk.

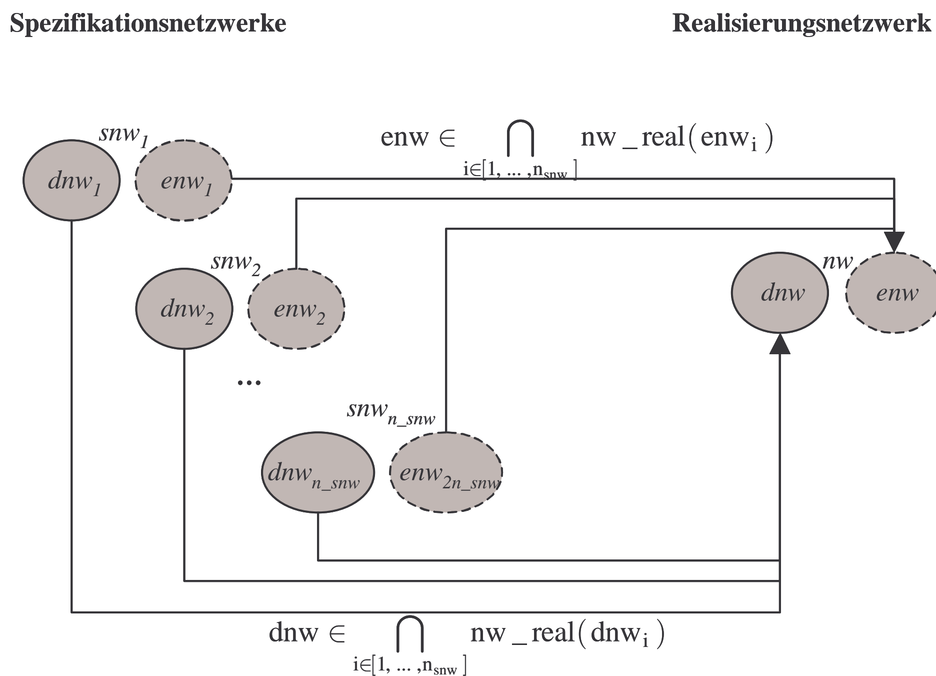


Abbildung 2.15: Spezifikationsnetzwerke und Realisierungsnetzwerk.

Mit dem Prädikat *topology* aus Definition 2.21 erfassen wir die topologische Aussage, die wir mit einer Topologiekomponente, in Abhängigkeit von ihrer topologischen Klassifikation, verbinden:

- **Verhaltenskomponente (behavior)**. Eine Verhaltenskomponente steht für einen Ausschnitt des Gesamtverhaltens. Mit dieser Art von Komponente verbinden wir keine Aussagen über die interne Netzwerkstruktur (Topologie) der Realisierung. Methodisch bedeutsam ist diese Komponentenart dann, wenn wir versuchen, die Komplexität des Gesamtsystems zu beherrschen, indem wir das Gesamte in weniger komplexe Teile dekomponieren (Divide et Impera), wobei die so erhaltene Topologie keinen bindenden Charakter für die Topologie einer Realisierung haben soll.

  Die Bedeutung der sogenannten *konzeptuellen Architektur* in [HNS99] entspricht einer Spezifikation mit Verhaltenskomponenten, da die topologi-

sche Struktur der konzeptuellen Sicht nicht bindend ist (für die Realisierung). Die Realisierung kann daher von der Struktur der konzeptuellen Sicht abweichen, etwa falls dies aus Qualitäts- oder Performanzanforderungen notwendig sein sollte.

Ebenso lassen sich die in FOCUS [BS00] vorgestellten zusammengesetzten Spezifikationen (*composite specifications*) in unserem Ansatz durch Spezifikationsnetzwerke abbilden, die aus behavior-Komponenten aufgebaut sind. In FOCUS werden in Form von Komponentennetzwerken gegebene Spezifikationen als composite specifications bezeichnet, wobei auf semantischer Ebene von der internen Struktur/Topologie der Spezifikation abstrahiert wird.

Durch Verhaltenskomponenten können wir beispielsweise in frühen Phasen einer Entwicklung deutlich machen, dass die Komponentenstruktur keinen bindenden Charakter für die Realisierung hat. Im Laufe einer Entwicklung, wenn sich die Komponentenstruktur auch aus Realisierungssicht als sinnvoll erweist, können wir dies dokumentieren, indem wir die Klassifikation von „behavior" zu „interface" oder zu „real" ändern.

- **Schnittstellenkomponente (interface)**. Analog zu einer Verhaltenskomponente beschreiben wir durch eine Schnittstellenkomponente eine (Teil-)Funktionalität. Jedoch verbinden wir damit zusätzlich eine Anforderung an die Aufteilung der Realisierung in Teilfunktionalitäten. In dieser Aufteilung muss die durch die Schnittstellenkomponente beschriebene Teilfunktionalität „identifizierbar" sein. Beispielsweise können wir mit Hilfe von Schnittstellenkomponenten die Einhaltung einer Schichtenarchitektur in der Realisierung fordern, das heißt die Aufteilung einer Gesamtfunktionalität in separierte, aufeinander aufbauende, Teilfunktionalitäten umfassende Schichten.

  Die oben genannte „Identifizierbarkeit" der durch eine Schnittstellenkomponente beschriebenen Teilfunktionalität in einer Realisierung, erfassen wir im Prädikat *topology* indem wir fordern, dass das Realisierungsnetzwerk ein Teilnetzwerk zu enthalten habe, zu dem die Schnittstellenkomponente eine Projektion darstellt. Das heißt, die syntaktische Schnittstelle der Schnittstellenkomponente ist ein Ausschnitt der Teilnetzwerkschnittstelle, und das Netzwerkverhalten bezüglich diesem Ausschnitt ist eine Verfeinerung des Verhaltens der Schnittstellenkomponente.

  Mittels Schnittstellenkomponenten können wir topologische Eigenschaften formulieren, ohne die Struktur der Realisierung exakt vorzugeben. So ist etwa die Zusammenfassung mehrerer Schnittstellenkomponenten (einer Spezifikation) in einer Komponente der Realisierung zulässig, so dass Schnittstellenkomponenten auch ein Mittel zur strukturierten Beschreibung von Realisierungskomponenten darstellen. In der Programmiersprache Java [Sun95] besteht ein ähnlicher Zusammenhang zwischen *Interface*- und *Class*-Elementen. Ein Class-Element realisiert im allgemeinen mehrere Interface-Elemente. Eine ähnliche Beziehung besteht zwischen einem *port* und der *computation* einer Komponente in der Architekturbeschreibungs-

sprache WRIGHT [All97][18].Da ein *port* in WRIGHT nicht nur Struktur, wie ein *interface* in Java, sondern auch (Teil-)Verhalten (einer Komponente) beschreibt, besteht eine enge Verwandtschaft zu unserem Konzept der Schnittstellenkomponente.

Umgekehrt lässt eine Schnittstellenkomponente auch eine weitergehende Dekomposition zu, so dass die Schnittstellenkomponente durch ein mehrelementiges Netzwerk realisiert wird. Beispielsweise kann eine Schicht (einer Schichtenarchitektur), repräsentiert durch eine Schnittstellenkomponente, durch ein Komponentennetzwerk realisiert werden. Damit sind Schnittstellenkomponenten ein Mittel zur hierarchischen Strukturierung von Netzwerken.

Methodisch von Bedeutung ist, dass wir durch interface-Komponenten neben der Unterspezifikation von Verhalten, auch topologische Unterspezifikation ermöglichen.

- **Realisierungskomponente (real)**. Durch eine Realisierungskomponente fordern wir, dass das Realisierungsnetzwerk eine Komponente mit derselben syntaktischen Schnittstelle und einem verfeinerten Verhalten enthält. Durch ein Netzwerk von Realisierungskomponenten können wir somit beispielsweise die Topologie der Realisierung exakt (1:1) festlegen.

  Realisierungskomponenten sind daher geeignet zur Repräsentation der in der jeweiligen Implementierung verwendeten Strukturierungseinheiten, zum Beispiel von COM- oder CORBA-Komponenten oder von den Objekten eines in einer objektorientierten Entwicklungsumgebung implementierten Softwaresystems.

  Die im V-Modell [BDI97] als *Module* bezeichneten Blätter der hierarchischen *Erzeugnisstruktur*, durch welche Implementierungseinheiten repräsentiert werden, lassen sich in unserem Ansatz durch Realisierungskomponenten geeignet abbilden.

Eine ähnliche Unterscheidung zwischen „syntaktischer Strukturierung, wie wir sie durch behavior Komponenten ermöglichen, und „semantischer Strukturierung“, wie sie durch interface und real Komponente ermöglicht wird, finden im Bereich der algebraischen Spezifikation in der Sprache CASL (common algebraic specification language) [ADH+02]. In dieser algebraischen Spezifikationssprache wird zwischen *Structured Specifications* und *Architectural Specifications* unterschieden:

> *„Architectural specifications are for describing the modular structure of software, in contrast to structured specifications where the structure is only for specification presentation purposes.”*
>
> [ADH+02]

Das folgende Beispiel 2.5 veranschaulicht den Zweck der topologischen Klassifikation in Spezifikationsnetzwerken.

[18] In [All97] wird der Zusammenhang zwischen den ports einer Komponente und der computation einer Komponente wie folgt charakterisiert: *„a computation carries out interactions described by the ports and shows how they are tied together.“*

Beispiel 2.5 (Topologische Bedeutung von Topologiekomponenten)

Um die Bedeutung der verschiedenen topologischen Komponentenarten zu verdeutlichen, betrachten wir die beiden in Abbildung 2.16 und Abbildung 2.17 skizzierten, aus einem Spezifikations- und einem Realisierungsnetzwerk bestehenden Paare.

In den beiden Abbildungen haben wir Komponenten der topologischen Klasse *behavior* durch *::B*, *interface* durch *::I* und *real* durch *::R* gekennzeichnet.

Das auf der linken Seite in Abbildung 2.16 dargestellte Spezifikationsnetzwerk umfasst die Komponenten *Benutzer-Schnittstelle* und *Anwendungslogik*. Da diese Komponenten als Verhaltenskomponenten klassifiziert sind, legen sie keine topologischen Eigenschaften fest. Daher ist die Zusammenfassung von Dialogsteuerungs- und Medien- beziehungsweise Leser-Verwaltungs-Funktionalität in jeweils einer Komponente zulässig. Das auf der rechten Seite in Abbildung 2.16 dargestellte Realisierungsnetzwerk hat diese Struktur.

Die Klassifizierung der Komponenten *MedienDB* und *LeserDB* als Schnittstellenkomponenten erlaubt es uns, beide Schnittstellen durch eine Komponente zu realisieren, jedoch müssen die Schnittstellen „erhalten" bleiben. Im Beispiel realisiert die Komponente *DB-Konnektor* die beiden Schnittstellen in Zusammenwirken mit der Komponente *DB*.

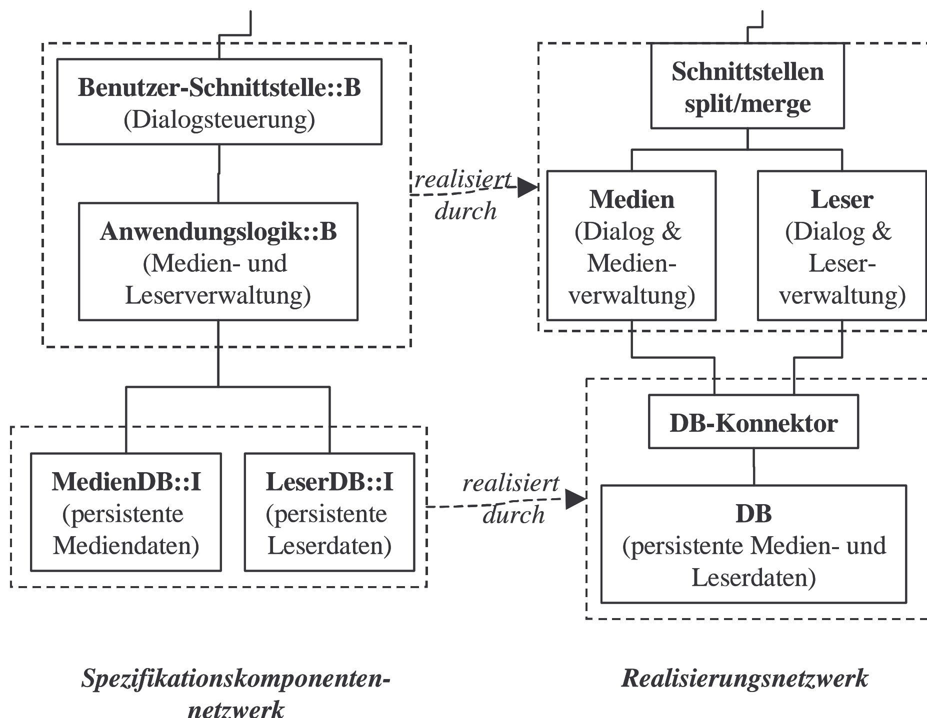


Abbildung 2.16: Topologisch typisierte Komponenten und ihre Realisierung

Ändern wir die topologische Klassifikation, wie in Abbildung 2.17 dargestellt, so ist die beschriebene Realisierung nicht mehr zulässig. Durch die Klassifizierung der *Benutzer-Schnittstelle* und der *Anwendungslogik* als Schnittstellenkomponenten

fordern wir eine entsprechend modulare Realisierung, in der die Dialogsteuerungs- von der Medien- beziehungsweise Leser-Verwaltungsfunktionalität separiert ist.

Allgemein können wir durch die Verwendung von Schnittstellenkomponenten die Einhaltung vorgegebener Schnittstellen in der Realisierung „erzwingen“, was für die Rolle von Architekturbeschreibungen als Implementierungsvorgaben bedeutsam ist.

Da wir in Abbildung 2.17 *MedienDB* und *LeserDB* als Realisierungskomponenten definiert haben, ist die Zusammenfassung beider Datenbanken in der Komponente *DB* nicht zulässig. Mit *DB-Konnektor* enthält die Realisierung zwar eine passende Schnittstelle, jedoch existiert weder zu *MedienDB* noch zu *LeserDB* ein (bis auf Isomorphie) strukturell äquivalentes Pendant. Damit wird deutlich, dass Komponenten vom Typ *real* geeignet sind, um beispielsweise Restriktionen durch einzubindende „Legacy“ Systemteile zu erfassen, in unserem Beispiel etwa zwei wiederzuverwendende, separate Datenbanken für Medien- und Leserdaten.

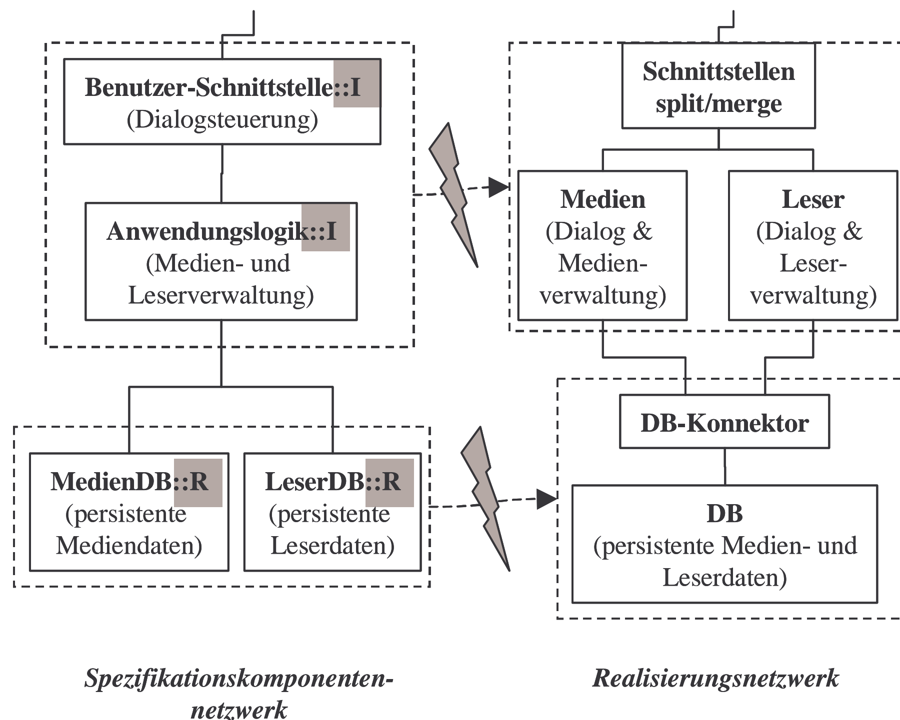


Abbildung 2.17: Ausschluss einer Realisierung durch topologische Typisierung

□

2.5.5 Komponentenautomaten

Wir führen nun Automaten als Mittel zur Beschreibung von Komponentenverhalten ein. Automaten haben ihre Ursprünge im Bereich der Sprachentheorie (siehe zum Beispiel [HU94]) um dann auch Verwendung für die (formale) Verhaltensspezifikation reaktiver Systeme zu finden. In einigen Ansätzen dienen Automaten (oder synonym Transitionssysteme) ausschließlich der Spezifikation von Sicherheitseigenschaften (zum Beispiel [Lam94], [Web91]). In anderen Ansätzen werden, wie in dieser Arbeit, durch Automaten sowohl Sicherheits- als auch Lebendigkeitseigenschaft formuliert (zum Beispiel in [Rum96]).

Automaten und die zugehörigen Notationen, zum Beispiel Statecharts [Har97], haben große praktische Relevanz. Dies macht die weite Verbreitung in pragmatischen Methoden, zum Beispiel Syntropy [CD94], OOTC [IBM97] und Catalysis [DW99a], sowie in dem Sprachstandard UML [OMG01] deutlich.

Im Unterschied zu einer axiomatischen Formulierung von Verhaltenseigenschaften mittels einer Logik, zum Beispiel prädikatenlogisch formulierte Spureigenschaften, stehen bei der Spezifikation mit Automaten nicht Eigenschaften des Gesamtverhaltens, sondern Eigenschaften einzelner Verhaltensschritte im Vordergrund. Ein Gesamtverhalten ergibt sich damit aus einer Folge von Verhaltensschritten. Diese schrittweise Verhaltenssicht hat einen konstruktiven Charakter, da die Teilschritte explizit spezifiziert werden, aus denen sich ein Gesamtverhalten zusammensetzt. Dadurch besteht eine konzeptuelle Nähe zu einer operationellen Verhaltensrealisierung. Daher eignen sich Automaten besonders für detaillierte Verhaltensbeschreibungen, die zum Beispiel im Feinentwurf als Vorgabe und schematisch umsetzbare Vorlage für die Implementierung sinnvoll eingesetzt werden können.

Ein weiterer Vorteil von Automaten ist die Existenz eines einfachen Kriteriums für die Vollständigkeit einer Spezifikation (in Form der Totalität der Transitionsrelation bezüglich einer Menge von Zuständen und Eingaben oder Aktionen, siehe unten). Daher eignen sich Automaten für vollständige Verhaltensspezifikationen.

Aus diesen Gründen wählen wir ein Automatenmodell, dass es erlaubt, Komponentenverhalten eindeutig zu definieren. Das bedeutet unter anderem, dass wir durch die gewählte Semantik von Automaten nicht nur Sicherheits-, sondern auch Lebendigkeitseigenschaften ausdrücken.

Neben dem operationellen Charakter von Automaten, aufgrund der schrittweisen Verhaltensbeschreibung, motiviert die Möglichkeit, durch Automaten eine Korrelation zwischen Zuständen und Ein-/Ausgabeverhalten einer Komponente herzustellen, den Einsatz von Automaten in unserem Ansatz. So kann ein Zusammenhang zwischen zustandsbezogenen Prozessereignissen und Komponenteninteraktionen formuliert werden. Durch einen Automaten können wir zum Beispiel darstellen, welche Information über den Datenzustand einer Komponente eine bestimmte Ausgabe(nachricht) enthält (vergleiche Beispiel 2.4).

Die grundlegenden Elemente eines Automaten sind Zustände und Transitionen.

Eine Transition steht für den Übergang von einem Ausgangs- in einen Zielzustand. Neben dem Zustandsübergang werden, in manchen Ansätzen, mit Transitionen bestimmte Ein- und Ausgaben assoziiert. In diesen Fällen ist eine bestimmte Transition möglich, falls der Automat in ihrem Ausgangszustand ist und zusätzlich die

Eingabe der Transition vorliegt. Zudem bedeutet die Ausführung einer Transition neben dem Zustandsübergang, dass eine entsprechende Ausgabe gemacht wird.

In unserem Ansatz nutzen wir Automaten zur Beschreibung von Komponentenverhalten, so dass wir unser Automatenmodell an unserem Komponentenbegriff ausrichten. Insbesondere dienen uns Automaten dazu, eine Korrelation zwischen dem (Daten-)Zustand einer Komponente und seinem Ein-/Ausgabeverhalten herzustellen. Damit ergeben sich folgende Eigenschaften für unser Automatenmodell:

- Da Komponenten Nachrichten empfangen und versenden, assoziieren wir mit jeder Transition eine bestimmte Ein- und Ausgabe. Damit besteht eine Ähnlichkeit zu den sogenannten Mealy-Automaten (siehe [HU94]).

- Die Ein- und Ausgaben einer Transition sind endliche Nachrichtensequenzen auf Ein- beziehungsweise Ausgabekanälen. Sie entsprechen damit der Ein- beziehungsweise Ausgabe, die eine Komponente innerhalb eines Taktes macht.

- Das gezeitete/getaktete Komponentenverhalten wird dadurch „erzeugt", dass pro Takt genau eine Transition ausgeführt wird:

  befindet sich eine Komponente zu einem bestimmten Takt in einem Zustand s und liegt in diesem Takt eine Eingabe i vor, dann wird (nichtdeterministisch) eine der Transitionen ausgewählt, die durch den Ausgangszustand s und die Eingabe i charakterisiert sind. Die Ausführung der gewählten Transition bedeutet, dass die Komponente im nächsten Takt im Zielzustand der Transition ist und die mit der Transition assoziierte Ausgabe macht.

  Abbildung 2.18 skizziert diesen Zusammenhang. Die als Pfeil dargestellte Transition hat Ausgangszustand *a* und Zielzustand *z* sowie Eingabe *i* und Ausgabe *o*. Die Ausführung der Transition „generiert" ein Verhalten, dass sich als Folge von Dreitupeln, bestehend aus Zustand, Ein- und Ausgabe, modellieren lässt. Die Eingabe *i´* in Abbildung 2.18 wird durch die Umgebung bestimmt, die Ausgabe *o´* durch die der betrachteten Transition Vorangegangene.

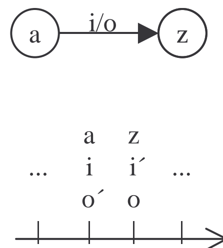


Abbildung 2.18: Automatentransition und „generierte" Ausführungsfolge

- Da eine Komponente, im allgemeinen, keinen Einfluss auf die Eingaben, das heißt das Umgebungsverhalten, hat, fordern wir, dass zu *jeder* Eingabehistorie ein Komponentenverhalten existiert. Für die Verhaltensfunktion einer Komponente bedeutet dies, dass wir deren Totalität fordern. Für den Automaten einer Komponente bedeutet es, dass in jedem Zustand zu jeder

möglichen Eingabe eine Transition existiert. Diese Forderung ist unter dem Begriff der „input enabledness“ bekannt (siehe zum Beispiel [LT89]).

Das oben beschriebene Automatenmodell legen wir in Definition 8.27 formal fest. Darauf aufbauend definieren wir das durch einen Automaten „generierte“ Verhalten:

- Definition 8.28 ordnet einem Automaten die „generierte“ Menge von Ausführungsspuren zu,
- Definition 8.29 eine (stromverarbeitende Verhaltens-)Funktion.

Im Systemmodell verwenden wir das genannte Automatenmodell zur Verhaltensbeschreibung von Topologiekomponenten. In der Automatensicht einer Komponente führen wir die hierzu notwendigen Systemmodellelemente ein:

Definition 2.22 (Automatensicht einer Topologiekomponente)

Jeder Komponente $k \in K$ ordnen wir eine Automatensicht zu, die aus folgenden Elementen besteht:

- eine nichtleere Menge von Initialelementen

  $\text{inits}_k \subseteq \text{states}_k \times \vec{O_k}^*$

- eine Transitionsfunktion

  $\delta_k : \left(\text{states}_k \times \vec{I_k}^*\right) \rightarrow \wp\left(\text{states}_k \times \vec{O_k}^*\right)$

  für die gilt:

  $\forall s \in \text{states}_k, i \in \vec{I_k}^*. \delta_k((s,i)) \neq \varnothing$.

  Damit schränken wir uns (im Systemmodell) auf *totale* Automaten ein, die somit keine Eingabe „blockieren“ und damit „input enabled“ sind (siehe Abschnitt 8.4.3).

□

Zusammen mit den Ein- und Ausgabekanälen der jeweiligen Topologiekomponente, bilden die Elemente der Automatensicht der Komponente einen Automaten, der die Ausführungsfolgen und die Verhaltensfunktion der Komponente bestimmt:

Definition 2.23 (Automatensicht und Komponentenverhalten)

Für jede Komponente $k \in K$ legen wir, mittels der Automatensicht, einen Automaten

$\text{STM}_k =_{\text{def}} (\text{states}_k, I_k, O_k, \delta_k, \text{inits}_k)$

gemäss Definition 8.27 fest. Durch diesen Automaten wird, entsprechend Definition 8.29, das Interaktionsverhalten behavior_k der Komponente k festgelegt:

$\text{behavior}_k =_{\text{def}} [\text{STM}_k]$

Die Ausführungsfolgen einer Komponente

$$\text{exec}_k \subseteq \left(\text{IS}_{I_k \cup O_k, \text{states}_k}\right)^\infty$$

definieren wir als die Ausführungsfolgen des Automaten der Komponente gemäss Definition 8.28:

$$\text{exec}_k =_{\text{def}} \text{executions}(\text{STM}_k)$$

□

Mit obiger Definition legen wir fest, dass die Verhaltensfunktion einer Topologiekomponente durch den Automaten der Komponente eindeutig bestimmt wird. Dies verdeutlicht, dass wir Automaten die Rolle der detaillierten Verhaltensbeschreibung von Komponenten zuweisen.

Die Forderung, dass die Automaten total zu sein haben, schließt nicht aus, dass wir in Entwicklungsprodukten Automaten verwenden, die nicht total sind (siehe Abschnitt 5.3).

2.5.6 Globaler Netzwerkzustand

Analog zu der Zustandsalgebra *FS* der ganzheitlichen Perspektive, durch die wir die Menge der fachlichen Zustände (in Form der Trägermenge der Sorte *State*) und den darauf definierten Operationen zusammenfassen, führen wir für jedes Spezifikationsnetzwerk eine Zustandsalgebra ein. Jede dieser Algebren umfasst die Sorte *State*, deren Trägermenge die Menge der Zustände des jeweiligen Spezifikationsnetzwerkes gemäss Definition 2.18 ist.

Die Zustandsalgebra eines Spezifikationsnetzwerkes dient dazu, anzugeben, auf welche Weise der fachliche Zustandsraum und die darauf definierten Operationen durch ein Spezifikationsnetzwerk realisiert werden. Wir gehen darauf detailliert in Abschnitt 2.6.1 ein, wo wir die Rolle von Spezifikationsnetzwerken als Bindeglieder zwischen

- fachlicher Aufgabenstellung, das heißt ganzheitlicher Perspektive mit expliziten Zuständen und den darauf definierten Operationen, und
- Realisierung der Aufgabenstellung, das heißt Realisierungsnetzwerk mit verborgenen, impliziten Zuständen, die sich nur mittelbar im Interaktionsverhalten widerspiegeln,

festlegen.

Neben der Zustandsalgebra geben wir, wieder analog zur ganzheitlichen Perspektive, die Klasse der (zulässigen) Zustandsalgebren an, um in Kapitel 3 zu unterscheiden zwischen exakten Spezifikationen und Spezifikationen, die nur einige aber nicht alle Eigenschaften der Zustandsalgebra angeben.

Definition 2.24 (Zustandssignatur und -algebra eines Spezifikationsnetzwerkes)

Globale, zustandsbezogene Eigenschaften eines Spezifikationsnetzwerkes snw_i, $i \in [1,...,n_{snw}]$, erfassen wir

- in der Zustandssignatur Σ_i
- der Klasse der Zustandsalgebren $C_i \subseteq Alg(\Sigma_i)$ sowie
- der Zustandsalgebra $KS_i \in C_i$.

Es gelte

- $State \in sorts(\Sigma_i)$
- $(KS_i)_{State} =_{def} states(snw_i)$.

  (vergleiche Definition 2.18)

□

2.5.7 Reaktionsabbildung

Geschäftsprozessspezifikationen sind wesentliche Elemente der Beschreibung einer fachlichen Aufgabenstellung von Informationssystemen. Sie geben im Wesentlichen an, wie in einem Anwendungssystem auf bestimmte Ereignisse reagiert wird (vergleiche Abschnitt 4.1). In der komponentenbasierten Perspektive erfassen wir auslösende Ereignisse und zugehörige Reaktionen in Form von (Mengen von) IS-Strömen. Für jedes Spezifikationsnetzwerk snw_i, $i \in [1,...,n_{snw}]$, ordnen wir Auslöser und Reaktionen einander durch die Abbildung

$$isreak_i : (IS_i)^* \rightarrow \wp\left((IS_i)^\omega\right)$$

zu. Ein Argument *X* von *isreak_i* verstehen wir als die verschiedenen Möglichkeiten, eine der Reaktionen aus *isreak_i(X)* auszulösen. Daher fordern wir in Definition 2.26, dass, falls

$isreak_i(X) = Y$

gilt, für jede Ausführungsfolge des betrachteten Spezifikationsnetzwerkes gilt:

auf jedes Auftreten eines Prozesses aus *X*, folgt nach endlicher Zeit ein Prozess aus *Y*.

Als Argumente lassen wir nur endliche Ströme zu, da unendliche Ströme keine Reaktion zur Folge haben können.

Nachdem wir die Realisierung von Prozessereignissen in Form von Mengen von IS-Strömen angeben (siehe Definition 2.28), sind die Abbildungen *isreak_i* die passenden Pendants zur Abbildung *reak* (vergleiche Definition 2.13) der ganzheitlichen Perspektive.

Definition 2.25 (Reaktionsabbildung der Ausführungsfolgen)

Bestimmte endliche Teilausführungsfolgen können andere Teilausführungsfolgen auslösen/zur Folge haben. Diesen Zusammenhang zwischen (Teil-)Ausführungsfolgen erfassen wir für jedes Spezifikationsnetzwerk snw_i, $i \in [1,...,n_{snw}]$, durch die Abbildung

$$\text{isreak}_i : (\text{IS}_i)^* \rightarrow \wp\left((\text{IS}_i)^\omega\right)$$

wobei wir folgende, abkürzende Schreibweise verwenden:

$$\text{IS}_i =_{\text{def}} \text{IS}_{\text{channels}(\text{snw}_i),\text{states}(\text{snw}_i)}.$$

□

Um unten die Interpretation von Reaktionsabbildungen im Sinne von Eigenschaften der Ausführungsfolgen von Spezifikationsnetzwerken formulieren zu können, definieren wir zwei Hilfsfunktionen *issometimes* und *isleadsto*, die für IS-Ströme Entsprechungen der temporallogischen Operatoren *sometimes* und *leadsto* (siehe gegebenenfalls [MP92]) sind.

Für die Funktion

$$\text{issometimes} : \text{IS}^\omega \times \text{IS}^\omega \rightarrow \mathbb{B}$$

gelte für alle $\psi, \varphi \in \text{IS}^\omega$:

$$\text{issometimes}(\psi, \varphi) \Leftrightarrow_{\text{def}} \exists i \in \mathbb{N}. \psi \sqsubseteq \varphi \uparrow i$$

Für die Funktion

$$\text{isleadsto} : \text{IS}^* \times \wp\left(\text{IS}^\omega\right) \times \text{IS}^\infty \rightarrow \mathbb{B}$$

gelte für alle $t \in \text{IS}^*, R \in \wp\left(\text{IS}^\omega\right), \varphi \in \text{IS}^\infty$:

$$\begin{aligned}\text{isleadsto}(t, R, \varphi) \Leftrightarrow_{\text{def}} \; & \forall i \in \mathbb{N}. \\ & t \sqsubseteq \varphi \uparrow i \Rightarrow \\ & \exists r \in R.\text{issometimes}(r, \varphi \uparrow (i + \#t))\end{aligned}$$

.

Definition 2.26 (Reaktionseigenschaften der Ausführungsfolgen von Spezifikationsnetzwerken)

Für jedes Spezifikationsnetzwerk snw_i, $i \in [1, ..., n_{\text{snw}}]$, legen wir durch die Abbildung *isreak$_i$* die Reaktionseigenschaften der Ausführungsfolgen des Netzwerkes fest. Es gelte:

$$\begin{aligned}& \forall t \in (\text{IS}_i)^*, R \in \wp\left((\text{IS}_i)^\omega\right), \varphi \in \text{exec}(\text{snw}_i). \\ & \qquad (\text{isreak}_i(t) = R) \Rightarrow \text{isleadsto}(t, R, \varphi)\end{aligned}$$

□

2.6 Perspektiven-Zusammenhänge

Neben der Beschreibung der fachlichen Aufgabenstellung sowie des Softwaresystems und dessen Umgebung ist die Beschreibung der Zusammenhänge zwischen Aufgabenstellung und dessen softwarebasierter Realisierung ein wichtiger Entwicklungsinhalt. Im Folgenden geben wir die hierzu notwendigen Systemmodellelemente an, die

- zum einen der konstruktiven Beschreibung davon dienen, auf welche Weise ein Realisierungsnetzwerk eine fachliche Aufgabenstellung realisiert, und
- zum anderen die Kriterien für die Beantwortung der Frage darstellen, wann wir ein Realisierungsnetzwerk als Realisierung der zugehörigen fachlichen Aufgabenstellung verstehen (analytischer Zweck).

Die unter dem Begriff Perspektiven-Zusammenhänge zusammengefassten Systemmodellelemente beziehen sich konkret darauf,

- wie durch die Zustandsräume der Komponenten von Netzwerken, der durch Entitäten aufgespannte, globale (fachliche) Zustandsraum realisiert wird, und
- durch welche Ausführungsfolgen von Komponenten(-netzwerken) Prozesse beziehungsweise Prozessereignisse realisiert werden.

Sowohl für das Verständnis dafür, auf welche Weise ein Softwaresystem eine fachliche Aufgabenstellung realisiert, als auch um den schrittweisen Übergang von der Spezifikation der fachlichen Aufgabenstellung hin zur Spezifikation der Realisierung dieser Aufgabenstellung durch ein Komponentennetzwerk (im Falle des Forward Engineering) oder umgekehrt (im Falle des Reverse Engineering) zu ermöglichen, müssen Entwicklungsprodukte und damit Systemmodellelemente zur Charakterisierung der Perspektivenzusammenhänge zur Verfügung stehen. Zum Beispiel muss beschrieben werden können, durch welche Komponenteninteraktionen eine fachliche Aktion, das heißt ein mit einer Aktion markiertes Prozessereignis, realisiert wird. So muss es beispielsweise möglich sein festzulegen, dass die Aktion „Leser über Verfügbarkeit des vorgemerkten Mediums benachrichtigen“ durch das Senden einer E-Mail-Nachricht, vom Bibliotheks-Softwaresystem an die, das E-Mail-Konto des Lesers verwaltende, Komponente, realisiert wird. Auch die Verteilung fachlicher Aufgaben auf das zu entwickelnde Softwaresystem und dessen Umgebung (Festlegung der Systemgrenze) halten wir durch Elemente der Perspektiven-Zusammenhänge fest, indem wir angeben, welche fachlichen Entitäten durch das Softwaresystem abgebildet werden und an welchen Prozessereignissen das Softwaresystem beteiligt ist (siehe Kapitel 6).

Die Bedeutung der Dokumentation der Zusammenhänge zwischen Aufgabenstellung und Realisierung wird deutlich, wenn aufgrund sich ändernder, fachlicher Anforderungen Änderungen des Softwaresystems notwendig werden. Allgemein verbessert die explizite Spezifikation der Zusammenhänge die Änderbarkeit und Wartbarkeit einer Softwarelösung.

Während die ganzheitliche Perspektive auf Modellierungskonzepten beruht, die geeignet sind zur Beschreibung einer fachlichen Aufgabenstellung, beruht die

komponentenbasierte Perspektive auf den Konzepten, die sich für die Beschreibung eines Softwaresystems und seiner Umgebung (und damit einer softwarebasierten Lösung einer Aufgabenstellung) eignen. Wir haben die Adäquatheit der verwendeten Modellierungskonzepte in den Abschnitten 2.3 und 2.5 motiviert. Da wir in den beiden Systemmodell-Perspektiven unterschiedliche Modellierungskonzepte verwenden, kann die Realisierungsbeziehung zwischen den Perspektiven nicht durch eine einfachen Verfeinerungsbegriff abgedeckt werden, in dem etwa die Mengeninklusion zwischen zwei Mengen *derselben* Elementarten, zum Beispiel Mengen unendlicher Zustandssequenzen, wie etwa in TLA [Lam94], gefordert und beschrieben wird. Vielmehr muss in unserem Ansatz ein Zusammenhang hergestellt werden, zwischen einer prozessbasierten Verhaltensmodellierung mit globalem Zustandsraum und einer Verhaltensmodellierung auf der Grundlage von durch Nachrichtenaustausch interagierenden, zustandskapselnden Komponenten.

Wir führen im Folgenden die Systemmodellelemente zur Charakterisierung der Perspektivenzusammenhänge ein, wobei wir zunächst Zustandsaspekte und anschließend daran Verhaltensaspekte betrachten.

2.6.1 Zustandsrealisierung

Ein Teil der Aufgabe von Informationssystemen ist es, fachliche Informationen zu verwalten. In der ganzheitlichen Perspektive beschreiben wir mit der Zustandsalgebra *FS* den durch Entitäten aufgespannten fachlichen Zustandsraum sowie dessen charakteristische Operationen. Durch Entitäten und ihre Zustände modellieren wir Informationen über physische oder ideelle Objekte der Anwendungswelt, zum Beispiel die Informationen, welche den Status eines Airbagsensors oder eines Kontos repräsentieren.

Da wir die ganzheitliche Perspektive eines Entwicklungssystems als fachliche Aufgabenstellung verstehen, fordern wir von dem Komponentennetzwerk, das durch das zu entwickelnde Softwaresystem und dessen Umgebung gebildet wird, dass es diese fachliche Information geeignet abbildet. Geeignet bedeutet hierbei, dass zu allen charakteristischen Operationen des fachlichen Zustandsraums Entsprechungen auf dem Zustandsraum des Netzwerks angegeben werden können. Dabei lassen wir (zunächst) unberücksichtigt, inwieweit die abzubildende Information relevant für das Systemverhalten im Sinne von Prozessen beziehungsweise Interaktionen ist. Zum Beispiel müssen in unserem Ansatz alle Entitäten abgebildet werden, auch diejenigen, die in keinem Prozess bearbeitet werden. Diese (starke) Forderung ist zum Beispiel im Hinblick auf die Erweiterbarkeit eines Softwaresystems sinnvoll.

Die Vorstellung, dass eine Realisierung die Informationen der Anwendungswelt so abzubilden hat, dass für alle zustandsbezogenen Operationen Entsprechungen angegeben werden können, finden wir auf ähnliche Weise in [Hoa72], eine der ersten Arbeiten zu dieser Fragestellung:

> *"In the development of programs by stepwise refinement, the programmer is encouraged to postpone the decision of the representation of his data until after he has designed his algorithm, and has expressed it as an "abstract" program operating on "abstract" data. He then chooses for*

> *the abstract data some convenient and efficient concrete representation […]; and finally programs the primitive operations required by his abstract program.”* **[Hoa72]**

In unserem Systemmodell modellieren wir, sowohl in der ganzheitlichen als auch in der komponentenbasierten Perspektive, die Zustandsaspekte durch Σ-Algebren. Dadurch steht uns das Konzept der Σ-Homomorphismen zur Modellierung der Realisierungsbeziehung zwischen den Zustandsalgebren beider Perspektiven zur Verfügung. Dem Konzept der Σ-Homomorphismen liegt die Idee zugrunde, dass sie die „Ähnlichkeit“ der Trägermengen zweier Algebren bezüglich einer Menge gegebener Operationen beschreiben (vergleiche Definition 8.5 und zum Beispiel [EKM+82] und [BMP+86]).

Die fachlichen Zustandsaspekte erfassen wir im Systemmodell mit der Zustandsalgebra *FS*, die Zustandsaspekte der Realisierung in den Zustandsalgebren KS_i, $i \in [1,...,n_{snw}]$, der Spezifikationsnetzwerke. Deshalb enthält unser Systemmodell für jedes Spezifikationsnetzwerk snw_i einen Σ_{fs}-Homomorphismus von KS_i nach *FS*, der angibt, auf welche Weise das Netzwerk den fachlichen Zustandsraum realisiert.

Von unterschiedlichen Bezeichnern in beiden Perspektiven und von „nicht-fachlichen“ Anteilen der Realisierung (KS_i) abstrahieren wir, indem wir die Zustandsrechenstrukturen (*FS* und KS_i) mittels einem Signaturmorphismus σ_i und einer diesbezüglich möglichen Reduktbildung „entkoppeln“:

Definition 2.27 (Morphismen der Zustandsalgebren)

Für jedes Spezifikationsnetzwerk snw_i, $i \in [1,...,n_{snw}]$, beschreiben wir den Zusammenhang zwischen der Σ_{fs}–Algebra FS (Definition 2.9) und der Σ_i–Algebra KS_i (Definition 2.24) durch folgende Systemelemente:

- Signaturmorphismus $\sigma_i : \Sigma_{fs} \to \Sigma_i$, mit $\sigma_i(\text{State}) =_{def} \text{State}$,
- Σ_{fs}-Homomorphismus $hs_i : (KS_i)|_{\sigma_i} \to FS$

$(KS_i)|_{\sigma_i}$ steht für das σ_i-Redukt von KS_i (vergleiche Definition 8.4)

□

Die methodische Bedeutung der Systemmodellelemente aus Definition 2.27 wird in Abschnitt 6.2 deutlich, wo wir Produktarten zur Festlegung der Grenze des Softwaresystems einführen. Dabei modellieren wir die Spezifikation des durch das Softwaresystem abzubildenden Anteils des fachlichen Zustandsraums als Aussage über einen Rechenstruktur-Homomorphismus.

Jeder Σ_{fs}-Homomorphismus hs_i gibt an, auf welche Weise die Trägermengen und Funktionen der ganzheitlichen Perspektive in dem Spezifikationsnetzwerk snw_i realisiert werden.

Zusammen mit Definition 2.24, in der wir zu jedem Spezifikationsnetzwerk snw_i eine Rechenstruktur KS_i mit

$$(KS_i)_{State} = states(snw_i)$$

einführen, stellen wir mit Definition 2.27 folgendes Konsistenzkriterium (für Entwicklungssysteme) auf:

> *„Zu jedem Spezifikationsnetzwerk snw_i muss eine Algebra KS_i angegeben werden können, mit*
>
> $$(KS_i)_{State} = states(snw_i)$$
>
> *so dass FS und KS_i homomorph sind, wobei $(KS_i)_{State}$ auf FS_{State} abgebildet wird."*

Das bedeutet, dass durch den Zustandsraum des Spezifikationsnetzwerkes alle in Form von FS gegebenen fachlichen Informationen realisiert werden.

Weiterhin ist zu beachten, dass wir, mit der geforderten Homomorphie von FS und KS_i, für die Bestimmung der „Ähnlichkeit" der Zustandsmengen FS_{State} und $(KS_i)_{State}$ alle zustandsbezogenen Markierungen (T und Z) von Prozessereignissen geeignet berücksichtigen, da die Funktionen aus T und Z Anwendungen der Funktionen aus FS sind. Damit ist sichergestellt, dass mit obigem Konsistenzkriterium alle für die Zustandsrealisierung relevanten Aspekte berücksichtigt sind.

Das folgende Beispiel zeigt die Realisierung (des Zustandsraums) einer fachlichen Entität durch (die Zustandsräume) zweier Komponenten. Damit verdeutlicht das Beispiel, dass eine Entität nicht notwendigerweise durch eine Komponente realisiert wird, sondern im allgemeinen durch mehrere Komponenten. Daraus folgt wiederum, dass eine auf einzelne Komponenten bezogene Formulierung der Realisierungsbeziehung nicht ausreicht, sondern eine Betrachtung des gesamten Netzwerkes notwendig ist. In unserem Systemmodell spiegelt sich diese Anforderungen darin wider, dass wir den fachlichen Zustandsraum zu dem Zustandsraum in Bezug setzen, der durch die Gesamtheit der Komponenten eines Spezifikationsnetzwerkes aufgespannt wird (vergleiche Definition 2.27 zusammen mit Definition 2.24).

Beispiel 2.6 (komponentenbasierte Realisierung einer fachlichen Entität)

Wir gehen von einem Entwicklungssystem aus, dessen fachlicher Zustandsraum durch eine Mediendatenbank-Entität gegeben sei. Unter einer Mediendatenbank verstehen wir hier eine (endliche) Menge von Medien, das heißt von Werten einer Sorte *Medium*. Die fachliche Mediendatenbank sei durch zwei Datenbank-Komponenten realisiert, die das Datenbanksystem einer Teil- und einer Hauptbibliothek einer Organisation, beispielsweise einer Universität, seien.

In Abbildung 2.19 sind die fachliche Entität sowie die Zustandsräume der Komponenten durch Objekte in der Syntax von UML-Objektdiagrammen dargestellt. Für die Darstellung der Komponenten *TB* und *HB* mit ihren Zustandsräumen verwenden wir die in [DW99] vorgestellte UML-Variante.

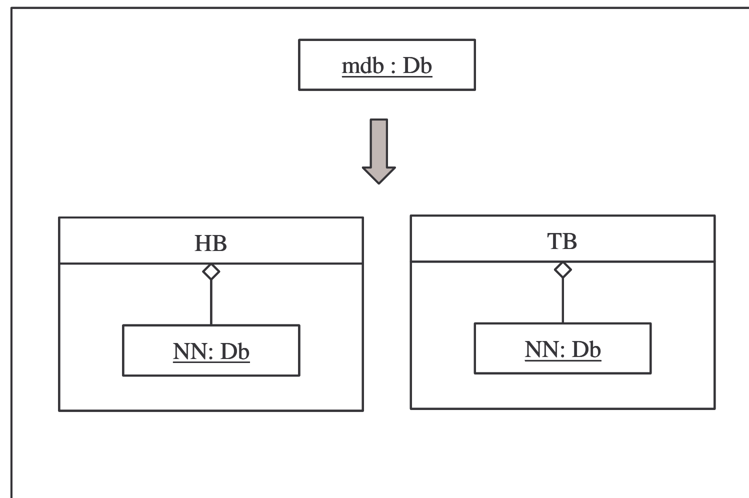


Abbildung 2.19: Realisierung einer fachlichen Entität.

Jedes Medium sei unter anderem dadurch charakterisiert, welchem Standort es zugeordnet ist. Die Teilbibliothekskomponente umfasse die Medien, welche dem Standort „Teilbibliothek“ zugeordnet sind, die Hauptbibliothekskomponente entsprechend Medien des Standortes „Hauptbibliothek“. Für jeden Netzwerkzustand, bestehend aus den Zuständen der beiden Datenbank-Komponenten, gelte damit, dass der realisierte fachliche Zustand, der Vereinigung der Medienmengen der beiden Komponenten entspricht.

Wir verdeutlichen die Beziehung zwischen fachlichem und Netzwerk-Zustandsraum, indem wir die Realisierung der Operation angeben, welche dem Einfügen eines Mediums in die fachliche Datenbank entspricht. Diese Operation wird realisiert durch eine Operation auf dem Netzwerkzustandsraum, die das einzufügende Medium, abhängig von dessen Standortwert, entweder der durch die Teil- oder die Hauptbibliothekskomponente gekapselten Medienmenge hinzufügt.

Die oben beschriebenen Eigenschaften des in diesem Beispiel angenommenen Entwicklungssystems, fassen wir in Begriffen des Systemmodells wie folgt zusammen (wir verzichten in diesem Beispiel auf die Subskription von Systemmodellelementen):

In der ganzheitliche Perspektive gelte:

Das System umfasse die Wertsorten

Db, Medium, Standort $\in$ V,

wobei Db die Sorte der endlichen Mengen von Werten der Sorte Medium sei.

Weiterhin seien die Operationen

insert, standort $\in \text{opns}(\Sigma_{fe})$ mit

insert : Db $\times$ Medium $\rightarrow$ Db

und

standort : Medium → Standort.

gegeben.

Der fachliche Zustandsraum sei durch eine einzige Datenbank-Entität aufgespannt:

$E = \{mdb\}$ mit $mdb \in id_{Db}$, so dass (wegen Definition 2.8)

$mdb.insert^* \in opns(\Sigma_{fs})$ mit $mdb.insert^* : State \times Medium \rightarrow State$.

Durch die Entität *mdb* modellieren wir die (Daten der) Mediendatenbank einer Bibliothek aus fachlicher Sicht.

Die Realisierung sei durch ein Spezifikationsnetzwerk snw_i beschrieben. Weiterhin nehmen wir den einfachen Fall an, dass die fachliche und die Netzwerkzustandssignatur übereinstimmen, das heißt $\Sigma_i = \Sigma_{fs}$ und σ_i die Identitätsabbildung ist, so dass insbesondere

$\sigma_i(mdb.insert^*) = mdb.insert^*$

gilt.

Die beiden Datenbank-Komponenten, aus denen das Netzwerk snw_i aufgebaut ist, nennen wir *TB* und *HB*, so dass gelte:

$\{TB, HB\} = K_i$

Der Zustandsraum der Komponenten entspreche jeweils dem einer (fachlichen) Mediendatenbank:

$states(TB) = states(HB) = A_{Db}$

Den Bezug zwischen fachlichem und Netzwerk-Zustandsraum legen wir mittels $(hs_i)_{State} : (KS_i)_{State} \rightarrow FS_{State}$

wie folgt fest:

$\forall s \in (KS_i)_{State}$.

$(hs_i)_{State}(s) = union^{KS_i}(s(TB), s(HB))$

wobei

$union : Db \times Db \rightarrow Db$ (aus Σ_i) die Operation der Mengenvereinigung bezeichne.

Die Realisierung der Funktion $(mdb.insert^*)^{FS}$ ist entsprechend dem Signaturmorphismus σ_i gegeben durch $(mdb.insert^*)^{KS_i}$. Für die (Realisierung der) Einfügeoperation gelte, dass Medien, deren Standort die Hauptbibliothek ist, zu der Komponente HB hinzugefügt werden, und analog dazu, Medien der Teilbibliothek der Komponente TB hinzugefügt werden. Für alle $s \in KS_{State}$, $m \in KS_{Medium}$ gelte daher:

$$(\text{mdb.insert}\,*)^{KS_i}(s,m) = \begin{cases} s\left[{}^{HB}\!/\!_{\text{insert}^A(s(HB),m)}\right], \\ \text{falls standort}(m)=\text{"hauptbib"} \\ \\ s\left[{}^{TB}\!/\!_{\text{insert}^A(s(TB),m)}\right], \text{sonst} \end{cases}$$

□

Beispiel 2.6 veranschaulicht den Unterschied zwischen dem Konzept der Entität als einer reinen Informations/Daten-Einheit und dem Konzept der Komponente als einer interagierenden, zustandskapselnden Einheit. Zudem macht das Beispiel deutlich, dass im allgemeinen (die Zustandsräume) mehrerer Komponenten eines Netzwerkes zur Realisierung einer Entität beitragen, so dass wir im Systemmodell für die Zustandsrealisierung nicht (nur) einzelne Komponenten, sondern ein gesamtes Netzwerk und dessen Zustandsraum zum fachlichen Zustandsraum in Bezug setzen müssen.

2.6.2 Verhaltensrealisierung

In der ganzheitliche Perspektive verwenden wir Prozesse (gemäss Definition 8.9) zur Verhaltensmodellierung und damit ein Modell, in dem Ereignisse (der Anwendungswelt) und die kausalen Abhängigkeiten zwischen Ereignissen in Form einer Partialordnung beschrieben werden. Die Partialität der Kausalordnung erlaubt es uns, die Unabhängigkeit von Ereignissen explizit zu erfassen.

Im Unterschied zur ganzheitlichen Perspektive basiert das Verhaltensmodell der komponentenbasierten Perspektive auf Strömen, also einem sequentiell strukturierten Modell (vergleiche Abschnitt 8.1.5). Als Elemente der Ströme verwenden wir Belegungen von Kanalbündeln/-mengen. Daher können wir, trotz der sequentiellen Grundstruktur, Parallelität in Form von gleichzeitiger Interaktion auf unterschiedlichen/parallelen Kanälen explizit machen. In Abbildung 2.20 veranschaulichen wir dies anhand eines Prozesses, dessen unabhängige, mit b und c markierte Ereignisse durch zwei gleichzeitige Interaktionen auf unterschiedlichen Kanälen realisiert werden (vereinfachend nehmen wir hier an, dass jedes Prozessereignis genau einer Komponenteninteraktion entspricht).

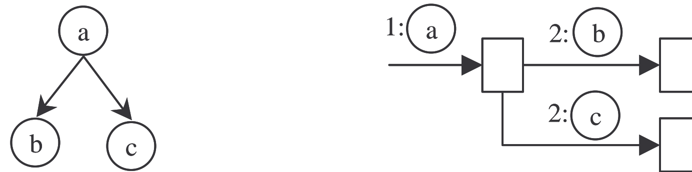


Abbildung 2.20: Unabhängige Prozessereignisse und parallele Komponenteninteraktion.

Um einen Bezug zwischen den Verhaltensmodellen der beiden Perspektiven des Systemmodells herstellen zu können, muss ein Übergang von einem Partialordnungsmodell zu einem sequentiell strukturierten Modell vollzogen werden. Hierfür gibt es im wesentlichen zwei Interpretationsmöglichkeiten von Prozessen (für eine detaillierte Diskussion siehe [GG98]):

- *Interleaving-Semantik*. Unabhängige Prozessereignisse werden in eine beliebige, (strikt) sequentielle Reihenfolge gebracht (Totalisierung der Partialordnung).
- *Step-Semantik*. Kausal unabhängige Ereignisse finden entweder parallel/gleichzeitig/im selben Takt statt, oder in beliebiger Reihenfolge.

Da wir bei dem in unserem Ansatz verwendeten Modell der Ausführungsfolgen von Komponenten Parallelität/Gleichzeitigkeit explizit erfassen können, wählen wir die Step-Semantik von Prozessen, für den Übergang von Prozessen zu einem strombasierten Modell.

Mathematisch bedeutet dies, dass wir einem Prozess mittels der Step-Semantik eine Menge von Strömen über Multimengen von Ereignismarkierungen zuordnen. Jede Multimenge eines Stroms steht dabei für die Ereignisse, die in einem Schritt stattfinden. Abbildung 2.21 gibt die Strommengen der Interleaving- sowie der Step-Semantik des bereits in Abbildung 2.20 dargestellten Prozesses an.

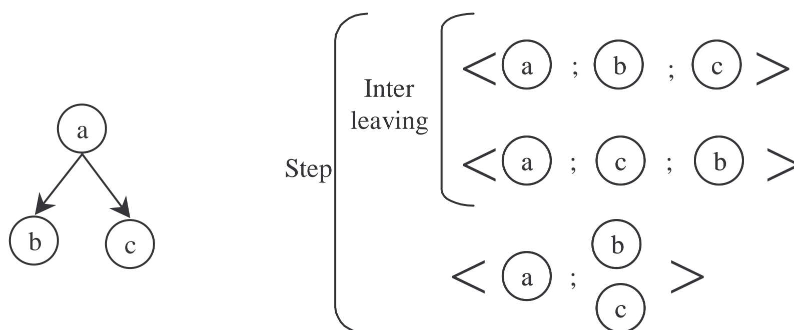


Abbildung 2.21: Interleaving- und Step-Semantik eines Prozesses.

Der Vorteil der Verwendung der Step-Semantik gegenüber der (reinen) Interleaving-Semantik ist, dass die Step-Semantik nicht von möglicher Parallelität/Gleichzeitigkeit im Falle von kausal unabhängigen Ereignissen abstrahiert und somit fein genug ist, um Überspezifikation zu vermeiden.

Mit der Step-Semantik gehen wir von Prozessen über zu Strömen über Multimengen von Ereignismarkierungen. Ausführungsfolgen von Komponenten und Komponentennetzwerken modellieren wir durch Ströme von Interaktions-Zustands-Tupeln (IS-Ströme, vergleiche Definition 8.22). Daher benötigen wir neben der Step-Semantik noch eine geeignete Korrelation von Ereignismarkierungen und IS-Tupeln beziehungsweise IS-Strömen, um Prozesse und Ausführungsfolgen korrelieren zu können. Abbildung 2.22 illustriert diese Zusammenhänge.

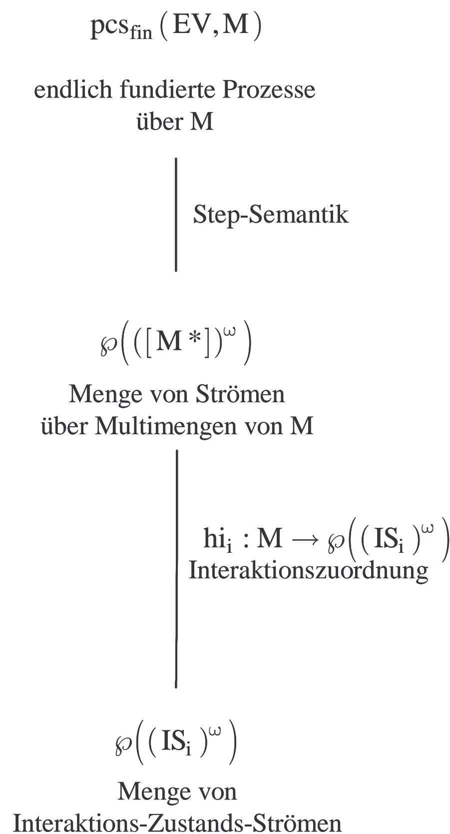


Abbildung 2.22: Zusammenhänge zwischen Prozessen und Ausführungsfolgen von Spezifikationsnetzwerken

Wir betrachten zunächst die Korrelation für die Markierungsart der Aktionsbezeichner *Act* eines Systems. Ein Beispiel hierfür ist die Markierung *vormerkungswunsch(m,l)*, wobei *m* eine Entitätsidentifikator für Medienentitäten und *l* ein Identifikator für Leserentitäten seien. Mit Aktionsbezeichnern assoziieren wir in der ganzheitlichen Perspektive keine weitere Bedeutung (im Unterschied zu den zustandsbezogenen Markierungsarten, siehe unten). Daher interpretieren wir Aktionsbezeichner in der komponentenbasierten Perspektive sehr allgemein als eine Menge von (Teil-)Ausführungsfolgen. Auf Notationsebene lässt sich diese Interpretation zum Beispiel anhand von Sequenzdiagrammen anschaulich darstellen: der Aktionsbezeichner benennt ein Sequenzdiagramm, durch welches die Menge der zugehörigen Ausführungsfolgen angegeben wird (siehe Beispiel 5.6 sowie [Krü01] für eine detaillierte Behandlung dieser Thematik).

Im Unterschied zu Aktionsbezeichnern, verbinden wir mit den zustandsbezogenen Ereignismarkierungen, T und Z, in der ganzheitlichen Perspektive eine bestimmte Bedeutung (vergleiche Definition 2.11):

- jede Markierung aus Z steht für ein Zustandsprädikat $FS_{State} \rightarrow Bool$
- jede Markierung aus T steht für eine Abbildung $FS_{State} \rightarrow FS_{State}$

Diese Bedeutung von zustandsbezogenen Markierungen bestimmt auch deren Interpretation/Realisierung in der komponentenbasierten Perspektive:

- Eine Markierung aus Z steht für eine Menge von Ausführungsfolgen, wobei in jeder dieser Ausführungsfolgen mindestens ein Zustand enthalten sein muss, der die Markierung erfüllt.
- Eine Markierung aus T steht für eine Menge von Ausführungsfolgen, wobei in jeder dieser Ausführungsfolgen mindestens eine Zustandsänderung gemäss der Markierung enthalten sein muss.

Mathematisch erfassen wir die Realisierung von Ereignismarkierungen durch Abbildungen

$$hi_i : M \rightarrow \wp\left((IS_i)^\omega\right)$$

für jedes Spezifikationsnetzwerk snw_i eines Entwicklungssystems. Für die Abbildungen hi_i legen wir im Systemmodell die oben skizzierten, grundlegenden, für die verschiedenen Markierungsarten spezifischen Eigenschaften fest:

Definition 2.28 (Interaktionszuordnung)

Für jedes Spezifikationsnetzwerk snw_i, $i \in [1,...,n_{snw}]$, legen wir mittels der Abbildung

$$hi_i : M \rightarrow \wp\left((IS_i)^\omega\right)$$

fest, durch welche (alternativen) IS-Ströme der komponentenbasierten Perspektive wir Ereignismarkierung interpretieren/realisieren. Für alle Markierungen $z \in Z$ und $t \in T$ fordern wir:

$$hi_i(z) \subseteq \left\{\varphi \in (IS_i)^\omega \,\middle|\, \exists n \in \mathbb{N}.z\big((hs_i)_{State}((\Pi_2(\varphi))@n)\big) = true^{FS}\right\}$$

$$hi_i(t) \subseteq \Big\{\varphi \in (IS_i)^\omega \,\Big|\, \exists n \in \mathbb{N}. \big((hs_i)_{State}(\Pi_2(\varphi)@n),(hs_i)_{State}(\Pi_2(\varphi)@(n+1))\big) \in R(t)\Big\}$$

Mit R(t) beziehen wir uns auf die durch t charakterisierte Zustandsrelation aus Definition 2.11.

□

Zu beachten ist, dass an dieser Stelle, der in Abschnitt 2.6.1 diskutierte Aspekt der Zustandsrealisierung einfließt, da die Anforderungen an die, ein Ereignis realisierenden, Ausführungsfolgen mittels des Homomorphismus hs_i formuliert werden.

Dadurch wird deutlich, dass mit den Ereignismarkierungen nur *fachliche* Zustandsaspekte einer Realisierung beschrieben werden.

Wie bereits erwähnt, verstehen wir die ganzheitliche Perspektive eines Systems als Beschreibung einer fachlichen Aufgabenstellung, die komponentenbasierte Perspektive als Beschreibung einer (komponentenbasierten) Realisierung dieser Aufgabenstellung. Für den Verhaltensaspekt bedeutet dies, dass wir die Prozessmenge *P* einer ganzheitlichen Perspektive im Sinne der *zulässigen Ausführungsfolgen* einer Realisierung, beschrieben in Form eines Spezifikationsnetzwerkes, zu interpretieren haben. Die Step-Semantik (aus Definition 8.16) zusammen mit der Interaktionszuordnung (aus Definition 2.28) sind die Bausteine für diese Interpretation der Prozessmenge P eines Systems. Wir halten dies mathematisch unten, in Definition 2.29 und Definition 2.30, fest.

Vorbereitend definieren wir die beiden Hilfsfunktionen *is_flatten* und *is_flatten**, durch die wir Multimengen von Ereignismarkierungen bezüglich einer gegebenen Interaktionszuordnung auf Interaktions-Zustands-Ströme abbilden.

Für eine Multimenge *f* und eine Interaktionszuordnung *h* liefert *is_flatten(f,h)* die Menge von IS-Strömen, die sich aus der Parallelkomposition (gemäss Definition 8.25) der Ströme ergeben, die *h* den Markierungen aus *f* zuordnet. Dies ist adäquat, da wir die mittels der Step-Semantik eines Prozesses definierten Multimengen als die Mengen „gleichzeitiger Ereignisse" verstehen und die Parallelkomposition gemäss Definition 8.25 die zu komponierenden Ströme taktweise gleichsetzt und komponiert.

Hilfsfunktionen (is_flatten und is_flatten*)

Sei X eine Zustandsmenge und Y eine Menge von Kanalbezeichnern.

Die Menge der Multimengen über (einer Markierungsmenge) M modellieren wir durch

$$(\mathrm{M} \rightarrow \mathbb{N}),$$

die leere Multimenge bezeichnen wir mit

$$\varnothing_{\mathrm{Multi}}$$

(für alle $m \in M$ gilt: $\varnothing_{\mathrm{Multi}}(m) = 0$), so dass

$$(\mathrm{M} \rightarrow \mathbb{N}) \backslash \varnothing_{\mathrm{Multi}}$$

für die Menge der nichtleeren Multimengen über M steht.

Um die Eigenschaften der Abbildung

$$\mathrm{is_flatten} : \Big(\big((\mathrm{M} \rightarrow \mathbb{N}) \setminus \varnothing_{\mathrm{Multi}} \big) \times \Big(\mathrm{M} \rightarrow \wp\big((\mathrm{IS}_{\mathrm{X,Y}})^{\omega} \big) \Big) \Big) \rightarrow \wp\big((\mathrm{IS}_{\mathrm{X,Y}})^{\omega} \big)$$

auf möglichst einfache Weise angeben zu können, verstehen wir Multimengen über M als (Repräsentanten von) Kongruenzklassen der Kongruenzrelation

$$\approx_{\#} \subseteq \mathrm{M}^* \times \mathrm{M}^*$$

die wir für alle $x, y \in M^*$ definieren durch:

$x \approx_{\#} y \Leftrightarrow_{def} \forall m \in M.\ \#(m©x) = \#(m©y)$.

Wir schreiben $[\varphi]_{\approx_{\#}}$, $\varphi \in M^*$, für die Kongruenzklasse (und damit die Multimenge), in der φ liegt. Für eine Multimenge f gilt

$$f = [\varphi]_{\approx_{\#}}$$

genau dann, wenn

$$\forall m \in M.\ \#(m©\varphi) = f(m).$$

Damit können wir die Abbildung *is_flatten* definieren durch

$$is_flatten\left([\langle m \rangle]_{\approx_{\#}}, h\right) =_{def} h(m)$$
$$is_flatten\left([\langle m_1, m_2, ..., m_n \rangle]_{\approx_{\#}}, h\right) =_{def} h(m_1) \sim h(m_2) \sim ... \sim h(m_n)$$

wobei ~ die Parallelkomposition von (Mengen von) IS-Strömen aus Definition 8.25 ist.

Wir erweitern *is_flatten*, für die Anwendung auf Ströme von Multimengen, zu

$$is_flatten^* : \left(((M \to \mathbb{N}) \setminus \varnothing_{Multi})^{\omega} \times \left(M \to \wp\left((IS_{X,Y})^{\omega}\right)\right)\right) \to \wp\left((IS_{X,Y})^{\omega}\right)$$

mit

$$is_flatten^*(\varepsilon, h) =_{def} \{\varepsilon\}$$
$$is_flatten^*(x \,\&\, xs, h) =_{def} \left\{r \in IS^{\omega} \mid ft(r) \in is_flatten(x, h) \wedge rt(r) \in is_flatten^*(xs, h)\right\}$$

□

Die Abbildungen *is_flatten* und *is_flatten** sind wohldefiniert, da ~ kommutativ ist.

In der folgenden Definition verwenden wir *is_flatten*, um aus der Prozessmenge einer ganzheitlichen Perspektive, mittels der Interaktionszuordnungsabbildungen hi_i, die zulässigen Ausführungsfolgen der Spezifikationsnetzwerke snw_i, und damit indirekt des Realisierungsnetzwerkes *nw*, abzuleiten:

Definition 2.29 (Ausführungsfolgen der Systemprozesse)

Für jedes Spezifikationsnetzwerk snw_i, $i \in [1, ..., n_{snw}]$, leiten wir mittels der in Definition 8.16 gegebenen Step-Semantik von Prozessen und der in Definition 2.28 festgelegten Abbildung hi_i, aus der Prozessmenge P (siehe Definition 2.12) die Menge

$$GStreams_i \subseteq (IS_i)^{\omega}$$

mit

$$\begin{aligned} \text{GStreams}_i =_{def} \Big\{ \varphi \in (\text{IS}_i)^\omega \Big| \exists p \in [\text{pcs}_{fin}(\text{EV}, \text{M})], \psi \in (\text{M} \rightarrow \mathbb{N})^\omega . \\ p \in P \wedge \\ \psi \in \text{cexecutions}(\text{Step}(p)) \wedge \\ \varphi \in \text{is_flatten}^*(\psi, \text{hi}_i) \Big\} \end{aligned}$$

ab.

□

Definition 2.30 (Prozesse und Ausführungsfolgen)

Wir verstehen *GStreams$_i$* als die Menge der *zulässigen* Ausführungsfolgen des Spezifikationsnetzwerkes *snw$_i$* und fordern daher:

$\text{exec}(\text{snw}_i) \subseteq \text{GStreams}_i$

für alle $i \in [1, ..., n_{snw}]$.

□

2.6.3 Verwandte Arbeiten

Der Übergang von globalen, komponentenunabhängigen Verhaltensbeschreibungen zu Verhaltensbeschreibungen, die an der Interaktion von Komponenten orientiert sind, wird unter anderem in [Bro89] und in [Web91] behandelt. In beiden Ansätzen findet der Übergang statt, indem Ereignisse der globalen Sicht als Einbeziehungsweise Ausgabeaktionen von Komponenten klassifiziert werden (Lokalisierung von Ereignissen/Aktionen). Unser Ansatz ist in zweifacher Hinsicht eine Verallgemeinerung davon. Zum einen, da in unserem Ansatz ein (globales) Ereignis nicht notwendigerweise genau einer Interaktion entsprechen muss, sondern einer Folge von Interaktionen entsprechen kann. Zum zweiten, weil eine *Menge* von Ausführungsfolgen angegeben werden, die wir (nichtdeterministisch) als alternative Realisierungen eines (globalen) Ereignisses verstehen. Methodisch sinnvoll ist diese Verallgemeinerung, da der Zweck der ganzheitlichen Perspektive die (lösungsunabhängige) Beschreibung einer fachlichen Aufgabenstellung ist. Da aber im allgemeinen die komponentenbasierte Realisierung eines Prozessereignisses von der Netzwerkstruktur abhängt, würde eine 1:1 Zuordnung von Prozessereignis und Komponenteninteraktion eine lösungsunabhängige Prozessspezifikation verhindern. Prozessereignisse wären im allgemeinen zu feingranular und damit Überspezifikation erzwungen.

Die in unserem Ansatz gewählte Interpretation von Prozessereignissen finden wir in ähnlicher Form, allerdings auf Ebene von Notationen, in [Krü01] wieder. Dort wird ein ähnlicher Zusammenhang zwischen sogenannten High Level Message Sequence Charts (HMSCs) und (einfachen) Sequenzdiagrammen (Message Sequence Charts (MSCs)) hergestellt. Die Bezeichner der „Knoten“ von HMSCs sind Referenzen/Bezeichner von Sequenzdiagrammen, so dass folgendes gilt:

“High Level MSCs depict alternative, repeated, and parallel interaction patterns by relating MSC references; this enables specification of „roadmaps“ through sets of MSCs.“ [**Krü01**]

3 Datenspezifikation

Wie in Abschnitt 2.1 ausführlich motiviert, halten wir die Beschreibung der fachlichen Aufgabenstellung, die mit Unterstützung eines Informationssystems zu realisieren ist, für eine der Kernaufgaben einer Softwareentwicklung. Anhand der fachlichen Aufgabenstellung wird die Rolle des zu entwickelnden Informationssystems für und seine Einbettung in den fachlichen Kontext, das Anwendungssystem, erfassbar. Dies ist etwa für ein Systemverständnis aus Anwendersicht unverzichtbar.

Als Aufgaben von Informationssystemen sehen wir, wie in Abschnitt 2.1 beschrieben, die Verwaltung fachlicher Information und die Unterstützung fachlicher Abläufe.

Um diese Aufgaben erfüllen zu können, müssen Objekte der Anwendungswelt, fachliche Entitäten, in Informationssystemen (in Form von Daten) abgebildet werden. Somit ist die Spezifikation fachlicher Entitäten und der auf sie anwendbaren Operationen Teil der Beschreibung einer fachlichen Aufgabenstellung.

In unserem Systemmodell erfassen wir fachliche Entitäten und ihre Charakteristika, wie in Abschnitt 2.3.1 ausführlich behandelt, in Form

- der (fachlichen) Entitätssignatur Σ_{fe},
- der Klasse der Entitätsrechenstrukturen C_f,
- der Entitätsrechenstruktur $A \in C_f$, sowie
- der Menge der Entitäten E.

Daher spezifizieren wir in unserem Ansatz fachliche Daten/Entitäten durch Entwicklungsprodukte, welche Eigenschaften der genannten Systemmodell-Elemente festlegen.

Im Verlauf einer Entwicklung benötigen wir unterschiedliche Arten von Entwicklungsprodukten für die Spezifikation fachlicher Entitäten. Zwei Unterscheidungsmerkmale sind hierbei von Bedeutung:

- Zum einen unterscheiden wir zwischen Produkten, die nur einen Teil der Sorten von Entitäten und ihren Operationen adressieren, und Produkten, welche die Gesamtheit der Sorten und Operationen festlegen.
- Zum zweiten unterscheiden wir danach, ob ein Produkt die adressierten Sorten und Operationen (aus fachlicher Sicht) vollständig spezifiziert, oder ob es nur bestimmte Eigenschaften angibt.

Diese Differenzierung von Produktarten halten wir für notwendig, da nur so die Rolle/der Zweck, der einem Entwicklungsprodukt (für den Entwicklungsprozess) zugedacht wird, ausreichend erfasst werden kann. Beispielsweise muss es für einen Softwareentwickler erkennbar sein, ob eine vorliegende Spezifikation von Sorten und Operationen, die zu implementieren sind, aus fachlicher Sicht vollständig ist oder nicht. Ebenso ist es etwa für die Projektplanung bedeutsam, ob eine vorliegende Spezifikation alle zu realisierenden Sorten und Operationen enthält oder nur einen Ausschnitt davon.

Entsprechend unserem Systemmodell bedeuten die beiden oben genannten Unterscheidungsmerkmale folgende Differenzierung von Produktarten:

- **Gesamte Signatur oder Teilsignatur**. Ein Produkt bezieht sich entweder auf alle Sorten und Operationen der Entitätssignatur Σ_{fe} (und auf die entsprechenden Elemente der Rechenstrukturen aus C_f) oder nur auf einen Ausschnitt davon.
- **exakte Festlegung von C_f oder Obermenge.** Durch ein Produkt werden entweder alle zu erfüllenden fachlichen Eigenschaften festgelegt, das heißt C_f wird exakt festgelegt, oder nur bestimmte, unbedingt zu erfüllende (Muss-)Eigenschaften, das heißt nur eine Obermenge von C_f.

Im Rest dieses Kapitels gehen wir zunächst auf die Produktarten ein, in denen Teilsignaturen behandelt werden, anschließend daran auf Produktarten, welche die gesamte Entitätssignatur betreffen. Darauf folgend führen wir Produktarten zur Spezifikation des, durch Entitäten aufgespannten, fachlichen Zustandsraums ein. Abbildung 3.1 gibt einen Überblick über die Produktarten zur fachlichen Daten- und Zustandsspezifikation[19]. Wir schließen das Kapitel ab, indem wir Notationen und Spezifikationstechniken diskutieren, die wir für die konkrete Repräsentation und für die Bearbeitung der behandelten Produktarten für geeignet halten.

[19] Die als *fachliche Zustandssicht* bezeichnete Produktart entspricht der Komposition von *exakter Spezifikation von Entitäten* und *Spezifikation des Initialzustandes*. Wir führen diese komponierte Produktart in Kapitel 6 ein.

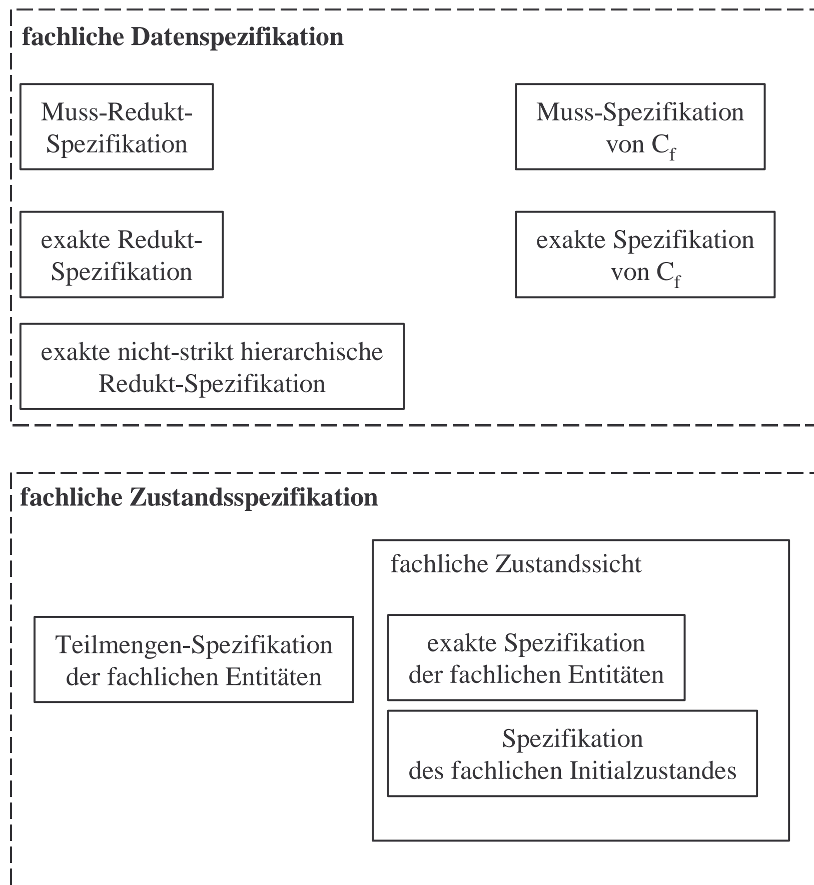


Abbildung 3.1: Produktarten zur fachlichen Daten- und Zustandsspezifikation

3.1 Reduktspezifikationen

Es bestehen im Wesentlichen zwei Gründe für die in diesem Abschnitt eingeführten Produktarten, die der Spezifikation von Sorten fachlicher Entitäten und darauf definierter Operationen dienen und mit denen wir keinen Anspruch auf Vollständigkeit bezüglich der betrachteten Sorten und Operationen verbinden:

Zum einen benötigen wir diese sogenannten Reduktspezifikationen, um Ergebnisse zu dokumentieren, die auf dem Weg zu einer umfassenden Spezifikation, welche alle Sorten und Operationen erfasst, entstehen. Zum Beispiel ist nur in seltenen Fällen am Anfang einer Entwicklung die Gesamtheit aller Arten fachlicher Entitäten und ihrer Charakteristika, das heißt ihrer Operationen, bekannt. Stattdessen werden diese schrittweise erarbeitet.

Zum zweiten dienen Reduktspezifikationen der strukturierten Dokumentation von Entitätssorten und Operationen. Beispielsweise lässt sich die Gesamtheit der fachlichen, abstrakten Datentypen einer Entwicklung auf strukturierte Weise spezifizieren, indem jeder Datentyp in jeweils einem entsprechenden Entwicklungsprodukt

spezifiziert wird. Die Produktart der *Analysis Class Descriptions* aus [IBM97] sind ein Beispiel hierfür:

> „Analysis Class Descriptions are summaries of all the information known about a class at the analysis level.
>
> [...]
>
> Analysis Class Descriptions are provided for two reasons:
>
> [...]
>
> To provide a single point of contact for analysis information regarding a particular class." ***[IBM97]***

Da wir in unserem Systemmodell die verschiedenen Entitätssorten und die darauf definierten Operationen in Form der Entitätsrechenstruktur *A* sowie deren fachliche Anforderungen in der Klasse C_f der (zulässigen) Entitätsrechenstrukturen erfassen (vergleiche Abschnitt 2.3), bedeutet die Spezifikation eines Ausschnitts der Sorten und Operationen, die Festlegung von Eigenschaften von *Redukten* von C_f (siehe Definition 8.4). Wir sprechen daher von Reduktspezifikationen der Entitätsrechenstrukturen.

Entwicklung bedeutet häufig schrittweise Verfeinerung/Vervollständigung spezifizierter Eigenschaften (vergleiche etwa [Wir71]). Daher müssen wir unterscheiden, zwischen Spezifikationen, die alle (relevanten) Eigenschaften von Redukten festlegen, und Spezifikationen, die nur einige Eigenschaften festlegen. Beispielsweise kann eine Spezifikation des abstrakten Datentypen der Medien (einer Bibliothek) noch bestimmte Fälle offen lassen, zum Beispiel, welchen Status ein Medium einnimmt, dass zurückgegeben wird und bereits vorgemerkt ist, um diese Fälle noch durch Rücksprache mit einem Experten des Anwendungsbereichs zu klären. Wir unterscheiden daher zwischen *muss-Redukt-Spezifikationen* und *exakten Redukt-Spezifikationen*. Muss-Redukt-Spezifikationen interpretieren wir so, dass sie nur eine Obermenge der (betrachteten Redukte der) zulässigen Entitätsrechenstrukturen C_f festlegen, exakte Spezifikationen hingegen legen (die jeweilige Reduktmenge von) C_f exakt fest.

Definition 3.1 (Muss-Spezifikation eines Reduktes der Entitätsrechenstrukturen)

Eine Muss-Spezifikation eines Reduktes der Entitätsrechenstrukturen eines Systems ist ein 3-Tupel $(X,\Sigma,C)_{muss_red}$ bestehend aus

- einer endlichen Menge X von Sorten
- einer Entitätssignatur[20] Σ bezüglich der Sortenmenge X und
- einer Menge $C \subseteq Alg(\Sigma)$.

Die Semantik definieren wir wie folgt:

[20] bezüglich dem Konzept der *Entitätssignatur* siehe Definition 8.8

$$\llbracket (X,\Sigma,C)_{muss_red} \rrbracket =_{def} \left\{ sys \in SYS \middle| X \subseteq V_{sys} \wedge \Sigma \subseteq (\Sigma_{fe})_{sys} \wedge (C_f)_{sys}\big|_{\Sigma} \subseteq C \right\}$$

□

Mit einer Muss-Spezifikation legen wir unbedingt zu erfüllende Anforderungen an die Menge C_f der (fachlich zulässigen) Entitätsrechenstrukturen fest. Eine Verstärkung dieser Anforderungen durch weitere Produkte ist zulässig.

Zu beachten ist, dass die Signatur Σ einer Muss-Redukt-Spezifikation $(X,\Sigma,C)_{muss_red}$ im allgemeinen nicht alle zur Erzeugung ihrer Sorten notwendigen Operationen umfasst. Dementsprechend ist für diese Art von Produkt die Menge C eine Teilmenge der Σ*-Algebren*, Alg(Σ), und nicht der Σ*-Rechenstrukturen*, Gen(Σ). Damit erfüllen auch solche Entwicklungssysteme die Spezifikation, deren Σ-Redukt von C_f nicht termerzeugt/erreichbar ist. Diese „offene“ Interpretation von Muss-Redukt-Spezifikationen ist methodisch wichtig, um etwa in weiteren Produkten zusätzliche (Generator-)Operationen auf sorts(Σ) angeben zu können.

Beispiel 3.1 (Muss-Redukt-Spezifikation)

Wir veranschaulichen den Zweck von Muss-Redukt-Spezifikationen anhand des folgenden Szenarios einer Systementwicklung:

Zu Beginn einer Anforderungsermittlung werden die Ausleihe und Rückgabe von Medien als Vorgänge des Anwendungsbereichs identifiziert. Unter Verwendung der algebraischen Spezifikationssprache SPECTRUM [BFG+93] werden Operationen und Eigenschaften der Sorte *Medium* festgelegt, die für die Ausleihe und Rückgabe von Medien benötigt werden.

```
MEDIUM = {

sort Medium, Leser, Status, id_Medium, id_Leser, id_Status, ;

mkons : Medium;
ausleihe : Medium × id_Leser → Medium;
rückgabe : Medium → Medium;

entleiher : Medium → id_Leser;
status : Medium → Status;

nn : → id_Leser;

entliehen : → Status;
verfügbar : → Status;

axioms ∀ m : Medium, l : id_Leser  in

status(mkons) = verfügbar;
entleiher(mkons) = nn;

(status(m) = entliehen) ⇒ ausleihe(m,l) = m;
```

```
(status(m) = verfügbar) ⇒ ((status(ausleihe(m,l) = entliehen) ∧
                    (entleiher(ausleihe(m,l)) = l))

(status(m) = verfügbar) ⇒ rückgabe(m) = m
(status(m) = entliehen) ⇒ ( (status(rückgabe(m)) = verfügbar) ∧
                    (entleiher(rückgabe(m)) = nn) )

endaxioms;

}
```

Abbildung 3.2: SPECTRUM-Spezifikation MEDIUM

Abbildung 3.2 zeigt die angesprochene SPECTRUM-Spezifikation. Sie trägt den Namen *MEDIUM* und führt die Sorten *Medium*, *Leser* und *Status* sowie die zugehörigen Identifikatorsorten ein, gefolgt von Generator- und Selektorfunktionen der Sorte *Medium* sowie einer Konstanten der Sorte id_{Leser} und zwei Konstanten der Sorte *Status*. Die, durch die Spezifikation festgelegte, Signatur bezeichnen wir mit Σ_{MEDIUM}. Die, mittels der losen Semantik der Sprache SPECTRUM, durch die Spezifikation beschriebene Menge von Σ_{MEDIUM}-Algebren (Klasse der Algebren, welche die Axiome erfüllen) bezeichnen wir mit C_{MEDIUM}.

Es wird davon ausgegangen, dass im Laufe der Anforderungsermittlung weitere Eigenschaften der betrachteten Sorten festzulegen sind. Zudem werden weitere fachliche Sorten und Operationen hinzu kommen. Daher wird der Spezifikation die Rolle einer *Muss-Redukt-Spezifikation* zugeordnet. Mit Σ_{MEDIUM} und C_{MEDIUM} bilden wir dementsprechend folgendes Produkt:

$(\{\text{Medium}, \text{Leser}, \text{Status}\}, \Sigma_{MEDIUM}, C_{MEDIUM})_{muss_red}$.

□

Definition 3.2 (Exakte Spezifikation eines Reduktes der Entitätsrechenstruktur)

Eine exakte Spezifikation eines Reduktes der Entitätsrechenstrukturen eines Systems ist ein 3-Tupel $(X, \Sigma, C)_{red}$ bestehend aus

- einer endlichen Menge X von Sorten
- einer Entitätssignatur Σ bezüglich der Sortenmenge X und
- einer Menge $C \subseteq Gen(\Sigma)$.

Die Semantik definieren wir wie folgt:

$$[\![(X, \Sigma, C)_{red}]\!] =_{def} \left\{ sys \in SYS \,\middle|\, X \subseteq V_{sys} \wedge \Sigma \subseteq (\Sigma_{fe})_{sys} \wedge (C_f)_{sys}\big|_{\Sigma} = C \right\}$$

□

Durch eine exakte Spezifikation legen wir eine als *vollständig* betrachtete Menge von Anforderungen/Eigenschaften bestimmter Sorten und Operationen fest. Unter Vollständigkeit verstehen wir, dass bezüglich dieser Sorten und Operationen die Formulierung stärkerer beziehungsweise zusätzlicher Anforderungen nicht zulässig ist, was aus der Semantikdefinition aufgrund der geforderten Äquivalenz

$$(C_f)_{sys}\big|_{\Sigma} = C$$

folgt. Diese Beziehung zwischen C_f und C, die wir durch eine exakte Reduktspezifikation festlegen, entspricht dem, was wir, aus dem Bereich der hierarchischen Konstruktion algebraischer Spezifikationen, als „persistent clientship" Relation kennen (vergleiche etwa [Bre91] und [WPP+83]):

Seien zwei Signaturen Σ, Σ' und zwei Spezifikationen $(\Sigma, C), (\Sigma', C')$ mit $C \subseteq Alg(\Sigma)$ und $C' \subseteq Alg(\Sigma')$ gegeben. Dann nennen wir

(Σ,C) *client* von (Σ',C')

falls gilt $\Sigma \subseteq \Sigma' \wedge C \subseteq C'|_{\Sigma}$.

(Σ,C) nennen wir *persistent client* von (Σ',C'), falls zusätzlich gilt:

$$C = C'|_{\Sigma}.$$

Im Falle der Termerzeugtheit von C bedeutet die *clientship* Beziehung, dass in C_f keine zusätzlichen Trägermengen-Elemente der primitiven Sorten eingeführt werden („no junk"). Die zusätzliche Eigenschaft der *Persistenz* fordert die Erhaltung der Äquivalenzen der primitiven Sorten („no confusion").

Im Unterschied zu exakten Redukt-Spezifikationen, fordern wir durch Muss-Redukt Spezifikation (vergleiche Definition 3.1) nur eine *clientship* Beziehung.

In Definition 3.2 findet sich der Aspekt der Vollständigkeit/Exaktheit auch in der Forderung

$C \subseteq Gen(\Sigma)$ (und nicht nur $C \subseteq Alg(\Sigma)$)

wieder. Diese Forderung ist zwar, aufgrund der im Rahmen der Semantik geforderten Äquivalenz von C_f und C (und der Systemmodell-Eigenschaft $C_f \subseteq Gen(\Sigma_{fe})$), redundant, macht diesen Zusammenhang aber explizit.

Bedeutsam ist die Eigenschaft der Exaktheit einer Spezifikation beispielsweise dann, wenn die Spezifikation als bindende Vorgabe für die Implementierung dienen soll: Durch den Ausschluss der Verstärkung von Anforderungen (an den betrachteten Signaturausschnitt) ist sichergestellt, dass es für die Implementierung ausreicht, die als exakt klassifizierte Spezifikation zu beachten, da weitere Spezifikationen keine der zulässigen Rechenstrukturen, und damit keine der möglichen Implementierungsalternativen (jede Rechenstruktur repräsentiert eine der möglichen Implementierungen), ausschließen können.

In jedem Fall spielt eine exakte Spezifikation eines Reduktes eine wichtige Rolle im Rahmen einer Entwicklungs*dokumentation*, und zwar als ein „hinreichender

Referenzpunkt“ für Informationen über die betreffenden Sorten und Operationen. Vergleiche hierzu obiges Zitat aus [IBM97] zu *Analysis Class Descriptions*.

Beispiel 3.2 (Exakte Reduktspezifikationen)

Wir setzen das Szenario aus Beispiel 3.1 fort, um die Bedeutung exakter Reduktspezifikationen zu verdeutlichen:

Im weiteren Verlauf der Anforderungsermittlung entstehen (exakte) Spezifikationen *LESER* und *STATUS* der Sorten *Leser* beziehungsweise *Status*. Zudem wird erkannt, dass neben der bisher betrachteten Ausleihe und Rückgabe von Medien, der Vorgang der Vormerkung von Medien zu berücksichtigen ist.

Daher wird die, in Abbildung 3.3 dargestellte, Spezifikation MEDIUM´ erstellt. Diese nimmt mit dem Ausdruck

enriches MEDIUM+LESER+STATUS+LIST

Bezug auf die bereits erstellten Spezifikationen und führt darauf aufbauend Operationen ein, die im Zusammenhang mit der Vormerkung von Medien notwendig sind:

```
MEDIUM´ = { enriches MEDIUM + LESER + STATUS + LIST;

-- Generatoren
vormerken : Medium ×id_Leser → Medium;
zuteilen : Medium → Medium;
zugeteilt : → Status;

-- Selektoren
vormerkungen : Medium → List id_Leser;

axioms ∀ m : Medium, l : id_Leser in

----vormerken bzgl. Selektorfunktionen
entleiher(vormerken(m,l)) = entleiher(m);
status(vormerken(m,l)) = status(m);

-- jeden Leser pro Medium hoechstens einmal vormerken
contains(vormerkungen(m),l) ⇒ vormerkungen(vormerken(m,l)) = vormerkungen(m)

not(contains(vormerkungen(m),l)) ⇒
vormerkungen(vormerken(m,l)) = listkons(l,vormerkungen(m))

----zuteilen bzgl. Selektorfunktionen
entleiher(zuteilen(m)) = entleiher(m)
status(zuteilen(m)) = zugeteilt
vormerkungen(zuteilen(m)) = vormerkungen(m)

----ausleihe wenn status = zugeteilt
(status(m) = zugeteilt ∧ l = last(vormerkungen(m))) ⇒
(entleiher(ausleihe(m,l)) = l ∧
status(ausleihe(m,l)) = entliehen ∧
```

```
vormerkungen(ausleihe(m,l)) = lrest(vormerkungen(m)))

(status(m) = zugeteilt ∧ not(l = last(vormerkungen(m))) ⇒
ausleihe(m,l) = m

----mkons bzgl. vormerkungen
vormerkungen(mkons) = emptylist;

----ausleihe bzgl. vormerkungen
(status(m) = verfügbar) ⇒ ((vormerkungen(ausleihe(m,l)) = vormerkungen(m))

----rückgabe bzgl. vormerkungen
(status(m) = entliehen) ⇒
(vormerkungen(rückgabe(m)) = vormerkungen(m))


endaxioms;

}
```

Abbildung 3.3: SPECTRUM-Spezifikation MEDIUM´.

Eine Analyse von MEDIUM´ durch Experten des Anwendungsbereichs ergibt, dass die Spezifikation (aus fachlicher Sicht) als vollständig zu bewerten ist. Dementsprechend wird ihr die Rolle einer *exakten Redukt-Spezifikation* zugeordnet:

$(X, \Sigma_{MEDIUM´}, C_{MEDIUM´})_{red}$ mit

$X =_{def} \{Medium, Leser, Status\}$

Zu beachten ist , dass jede Änderung der Eigenschaften, die mit einer exakten Reduktspezifikation festgelegt wurden, auch eine Verstärkung der Eigenschaften, nicht zulässig ist:

Sei $(X, \Sigma_{MEDIUM}, C)_{*_red}$ eine exakte oder eine Muss-Reduktspezifikation mit

$C \subseteq C_{MEDIUM´}$

so ist die Semantik der Produktmenge

$\{(X, \Sigma_{MEDIUM´}, C_{MEDIUM´})_{red}, (X, \Sigma_{MEDIUM´}, C)_{*_red}\}$

die leere (System-)Menge (vergleiche Definition 2.4).

Jedoch ist die Einführung zusätzlicher Operationen, relativ zu einer exakten Spezifikation, möglich, solange alle Term-Äquivalenzen erhalten bleiben. Beispielsweise können wir eine zusätzliche Operation (welche die Rückgabe und Vormerkung eines Mediums für ein und denselben Leser zu einer Operation zusammenfasst)

rück_vorm : Medium $\times$ id_{Leser} $\rightarrow$ Medium

mit

$\forall$ m : Medium, l : id_{Leser}:

rück_vorm(m,l) = vormerken(rückgabe(m),l)

ergänzen, da diese vollständig auf den existierenden Operationen *rückgabe* und *vormerken* abgestützt ist. Aus methodischer Sicht sind solche zusätzlichen, die Termäquivalenzen erhaltenden Operationen ähnlich zu den „Komfortfunktionen“, die in [Pae98a] als *work-specific system functions* bezeichnet werden, und dort in der Entwicklung von Use Cases gesondert behandelt werden.

Im Unterschied zu *rück_vorm*, wäre die Hinzunahme einer Funktion (welche die Rückgabe eines Mediums rückgängig macht)

rev_rückgabe : Medium → Medium

mit

rev_rückgabe(rückgabe(m)) = m

unzulässig, denn damit müssten die Terme

rückgabe(ausleihe(m,l)) und *rückgabe(ausleihe(m,l´))*

durch unterschiedliche Elemente der Trägermenge der Sorte *Medium* interpretiert werden, falls $l \neq l'$, denn zum Beispiel macht die Selektorfunktion

status : Medium → Status

die Unterscheidung von

ausleihe(m,l) und *ausleihe(m,l´)* (falls $l \neq l'$)

notwendig. Dies ist ohne *rev_rückgabe* nicht notwendig. Die Notwendigkeit der Unterscheidung wird klar, wenn wir eine zustandsbasierte Realisierung der Sorte *Medium*, beispielsweise durch eine objektorientierte Klasse, betrachten:

> Um durch die Anwendung von rev_rückgabe nach einer Rückgabe eines Mediums auf den Zustand zurücksetzen zu können, den das Medium vor der Rückgabe hatte, müssen wir Information über den jeweils letzten Entleiher speichern.

rev_rückgabe kompensiert die „Wirkung“ der Funktion *rückgabe*. Bedeutsam sind solche „kompensierenden Funktionen“ beispielsweise für die Realisierung von „Compensating Action“ Transaktionen (siehe etwa [LSW01], [GS87]). Durch Gleichungsaxiome, wie oben für *rev_rückgabe*, lassen sich Funktion und zugehörige „Kompensationsfunktion“ auf klar erkennbare Weise zueinander in Bezug setzen.

□

Neben der Muss- und der exakten Reduktspezifikation benötigen wir die folgende Art von Spezifikation, die sogenannte *exakte, nicht strikt hierarchische Reduktspezifikation*. Diese Entwicklungsproduktart ist für den Fall bestimmt, wenn wir bestimmte Sorten und Operationen basierend auf, als gegeben angenommenen, Sorten und Operationen exakt spezifizieren. Diese als gegeben angenommenen „Basiselemente“ sind zwar (notwendigerweise) Bestandteile der Signatur, werden aber im allgemeinen innerhalb der Spezifikation nicht exakt festgelegt, im Unterschied zu den darauf aufbauenden Sorten und Operationen. Das bedeutet, dass wir zwar die *Signaturen* hierarchisch *strukturieren*, die zugehörigen Rechenstrukturen

aber nicht streng hierarchisch *spezifizieren*, da Basiselemente gar nicht oder nur unvollständig spezifiziert werden. Auf Notationsebene finden wir diesen Zusammenhang häufig in Ausdrücken der Form

import <Basiselemente>

wieder.

Ein Beispiel für die Notwendigkeit der nicht-strikt hierarchischen Spezifikation, ist die Spezifikation verschränkt rekursiver (abstrakter) Datentypen, wenn wir die beteiligten Datentypen in separaten Produkten erfassen wollen.

Definition 3.3 (exakte, nicht strikt hierarchische Redukt-Spezifikation)

Eine exakte, nicht strikt-hierarchische Redukt-Spezifikation ist ein 4-Tupel $(X,\Sigma,\Sigma',C)_{nhier_red_fe}$ bestehend aus

- einer endlichen Menge X von Sorten
- einer Entitätssignatur Σ bezüglich der Sortenmenge X
- einer Teilsignatur $\Sigma' \subseteq \Sigma$
- einer Menge $C \subseteq Alg(\Sigma)$, so dass für alle $c \in C, s \in sorts(\Sigma) \setminus sorts(\Sigma')$ gilt:

  s ist *erreichbar* durch $opns(\Sigma) \setminus opns(\Sigma')$.

Die Semantik dieser Art von Spezifikation definieren wir durch:

$$\llbracket (X,\Sigma,\Sigma',C)_{nhier_red_fe} \rrbracket =_{def} \{sys \in SYS \mid X \subseteq V_{sys} \wedge \\ \Sigma \subseteq (\Sigma_{fe})_{sys} \wedge \\ (C_f)_{sys}\big|_{\Sigma} \subseteq C \wedge \\ \forall c \in C. \left(c|_{\Sigma'} \in (C_f)_{sys}\big|_{\Sigma'}\right) \Rightarrow \left(c \in (C_f)_{sys}\big|_{\Sigma}\right)\}$$

□

Die Implikation

$$\left(c|_{\Sigma'} \in (C_f)_{sys}\big|_{\Sigma'}\right) \Rightarrow \left(c \in (C_f)_{sys}\big|_{\Sigma}\right)$$

in obiger Definition bedeutet, dass weitere Spezifikationen höchstens bezüglich Σ' zusätzliche Anforderungen stellen können, nicht aber bezüglich $\Sigma \setminus \Sigma'$. Dadurch und durch die Forderung, dass alle $s \in sorts(\Sigma) \setminus sorts(\Sigma')$ durch $opns(\Sigma) \setminus opns(\Sigma')$ erreichbar seien, drücken wir die Exaktheit der Spezifikation von $\Sigma \setminus \Sigma'$ im Rahmen einer nicht-strikt-hierarchischen Reduktspezifikation aus.

Die algebraische Spezifikationssprache CASL besitzt mit den sogenannten *functional units* ein Konzept, das ähnlich ist zu unseren nicht-strikt-hierarchischen Spezifikationen. Eine functional unit steht für eine (persistente) Funktion, die (ein Tupel

von) Klassen von Algebren, die Parameter der functional unit, auf eine Klasse von Algebren abbildet (vergleiche etwa [ADH+02]).

Beispiel 3.3 (nicht strikt hierarchische Reduktspezifikation)

Alternativ zu der, in Beispiel 3.2 beschriebenen, Fortsetzung des Entwicklungsszenarios, ist auch folgendes Vorgehen vorstellbar: Die Spezifikation *MEDIUM* wird zu Spezifikation *MEDIUM´´* aus Abbildung 3.4 erweitert. Dabei werden Eigenschaften der Sorten *Leser* und *Status* nur insoweit festgelegt, wie dies für die Eigenschaften der Sorte *Medium* von Bedeutung ist.

```
MEDIUM´´ = { enriches MEDIUM;

-- Signaturanteil der Sorte Leser
...

-- Signaturanteil der Sorte Status
zugeteilt : → Status;
...

-- Signaturanteil der Sorte List
...


-- Generatoren
vormerken : Medium × id_Leser → Medium;
zuteilen : Medium → Medium;

-- Selektoren
vormerkungen : Medium  → List id_Leser;


Status freely generated by entliehen, verfügbar, zugeteilt

axioms ∀ m : Medium, l : id_Leser in

---- wie in Medium´
...

endaxioms;

}
```

Abbildung 3.4: SPECTRUM-Spezifikation MEDIUM´´

Die in *MEDIUM´´* für die Sorte *Medium* und ihre charakteristischen Operationen festgelegten Eigenschaften werden als (fachlich) vollständig bewertet.

Im Unterschied zu Beispiel 3.2 gehen wir hier davon aus, dass nicht auf existierende, exakte Spezifikationen der „Basissorten", *Leser* und *Status*, zurückgegriffen werden kann (in Beispiel 3.2 auf syntaktischer Ebene durch Einbindung der Spezifikationen *LESER*, *STATUS* und *LIST* mittels *enriches LESER, STATUS, LIST* repräsentiert). Stattdessen gehen wir davon aus, dass Eigenschaften der „Basissorten" von anderer Seite und/oder zu einem späteren Zeitpunkt festgelegt werden. Dem-

nach wird die Spezifikation *MEDIUM´´* als *nicht strikt hierarchische, exakte Reduktspezifikation* eingestuft:

$(\{\text{Medium}, \text{Leser}, \text{Status}\}, \Sigma_{MEDIUM''}, \Sigma', C_{MEDIUM''})_{nhier_red}$

mit

$\Sigma' \subseteq \Sigma_{MEDIUM''}$ mit $\Sigma' = \Sigma_{Leser} \cup \Sigma_{Status} \cup \Sigma_{List}$

wobei Σ_{Leser} , Σ_{Status} ,und Σ_{List} die Signaturen, nicht die Spezifikationen, der betreffenden Sorten bezeichnen. Beispielsweise gilt:

sorts(Σ_{Status}) = {Status} und opns(Σ_{Status}) = {entliehen, verfügbar, zugeteilt}.

□

3.2 Umfassende Spezifikationen

Ergänzend zu Spezifikationen, die nur einen Ausschnitt der Sorten und Operationen fachlicher Entitäten (einer Entwicklung) adressieren, benötigen wir Produktarten, welche die Gesamtheit der Entitätssorten und –Operationen adressieren und festlegen. Beispielsweise muss ein Produkt, das dazu dient, die fachliche Aufgabenstellung einer Entwicklung *vollständig* zu dokumentieren, diese Eigenschaft besitzen, so dass garantiert ist, dass kein weiteres Produkt der betrachteten Entwicklung zusätzliche Sorten oder Operationen fachlicher Entitäten einführt. Vollständigkeit ist zudem eine typische Anforderung an Produktarten, die als Ziel/Endergebnis von Entwicklungsphasen definiert werden.

Bezüglich unseres Systemmodells bedeutet die Forderung, dass ein Entwicklungsprodukt alle Sorten und Operationen fachlicher Entitäten angibt, dass die Entitätssignatur Σ_{fe} eindeutig festgelegt wird. Analog zu den in Abschnitt 3.1 eingeführten Reduktspezifikationen, unterscheiden wir zwischen Produktarten, die für die Gesamtheit der Sorten und Operationen nur bestimmte beziehungsweise alle fachlich relevanten Eigenschaften festlegen. Das heißt, wir unterscheiden zwischen muss-Spezifikationen und exakten Spezifikationen.

Definition 3.4 (Muss-Spezifikation der Entitätsrechenstrukturen)

Eine Muss-Spezifikation der Entitätsrechenstrukturen eines Systems ist ein 3-Tupel $(X, \Sigma, C)_{muss_fe}$ bestehend aus

- einer endlichen Sortenmenge X
- einer Entitätssignatur Σ bezüglich der Sortenmenge X und
- einer Menge $C \subseteq Gen(\Sigma)$.

Die Semantik definieren wir wie folgt:

$$[\![(X,\Sigma,C)_{muss_fe}]\!] =_{def} \{ sys \in SYS \mid X \subseteq V_{sys} \wedge \Sigma = (\Sigma_{fe})_{sys} \wedge (C_f)_{sys} \subseteq C \}$$

□

Definition 3.5 (Exakte Spezifikation der Entitätsrechenstrukturen)

Die exakte Spezifikation der Entitätsrechenstrukturen eines Systems ist ein 2-Tupel $(X,\Sigma,C)_{fe}$ bestehend aus

- einer endlichen Sortenmenge X
- einer Entitätssignatur Σ bezüglich der Sortenmenge X und
- einer Menge $C \subseteq Gen(\Sigma)$.

Die Semantik definieren wir wie folgt:

$$[\![(X,\Sigma,C)_{fe}]\!] =_{def} \{ sys \in SYS \mid X \subseteq V_{sys} \wedge \Sigma = (\Sigma_{fe})_{sys} \wedge (C_f)_{sys} = C \}$$

□

Mittels umfassender Spezifikationen legen wir die Signatur Σ_{fe} eines Systems eindeutig fest. Die Hinzunahme von Sorten und Operationen wird dadurch ausgeschlossen. Damit sind umfassende Spezifikationen (innerhalb einer Dokumentation) dazu geeignet, einen „hinreichenden“ Überblick über die Gesamtheit der fachlichen Entitätssorten und –Operationen zu geben.

Für eine exakte Spezifikation der Entitätstrechenstrukturen gilt zusätzlich, dass sie die Voraussetzung ist, für die Formulierung *fachlicher* Anforderungen, hinsichtlich der Zustandsrealisierung, an das zu entwickelnde Informationssystem :

> *Nur wenn alle Eigenschaften fachlicher Entitäten bekannt sind ist klar, was es bedeutet, wenn wir von einem Informationssystem fordern, dass es bestimmte fachliche Entitäten zu realisieren hat.*

Aus diesem Grund legen wir in Definition 6.7 fest, dass eine exakte Spezifikation der Entitätsrechenstrukturen Bestandteil einer Anforderungsspezifikation ist.

3.3 Entitätsspezifikation

Ein Teil der fachlichen Aufgabenstellung einer Systementwicklung ist der zu realisierende fachliche Zustandsraum. In unserem Systemmodell spannen wir den fachlichen Zustandsraum durch eine endliche Menge *E* von Entitäten auf. Ein Zustand des Anwendungssystems setzt sich (aus fachlicher Sicht) aus den Zuständen der Entitäten zusammen (vergleiche Abschnitt 2.3).

Die in den Abschnitten 3.1 und 3.2 eingeführten Arten von Entwicklungsprodukten dienen der Festlegung von Eigenschaften der unterschiedlichen Arten von Entitäten; zum Beispiel können wir durch eine (Trägermenge der) Wertsorte *Medium* die

möglichen Zustände von Medien-Entitäten einer Bibliothek festlegen (vergleiche Abschnitt 2.3.1). Im Unterschied dazu, dienen die folgenden Produktarten der Festlegung der Entitäten, welche den fachlichen Zustandsraum bilden, das heißt der Entitätsmenge *E* eines Entwicklungssystems. Wir unterscheiden hierbei die Produktart zur Spezifikation eines Ausschnittes von derjenigen zur exakten Spezifikation der Entitätsmenge.

Definition 3.6 (Teilmengen- und exakte Entitätsspezifikation)

Eine Spezifikation der Entitätsmenge eines Systems besteht aus

- einer endlichen Menge X.

Die Semantik einer *Teilmengen-* (E⊆) und einer *exakten* (E) *Entitätsspezifikation* definieren wir wie folgt:

- $[\![(X)_{E\subseteq}]\!] =_{def} \{sys \in SYS \mid X \subseteq E_{sys}\}$
- $[\![(X)_{E}]\!] =_{def} \{sys \in SYS \mid X = E_{sys}\}$

□

Mit dem Initialzustand des Systemmodells erfassen wir die Zustände, welche die Entitäten zum „Systemstart" einnehmen. Die folgende Produktart dient der Festlegung des Initialzustandes.

Definition 3.7 (Spezifikation des Initialzustands)

Eine Spezifikation des Initialzustands besteht aus

- einer Abbildung $s_0 : E \to VAL$, wobei E eine endliche Menge und VAL eine nichtleere Menge sei.

Die Semantik definieren wir wie folgt:

$$[\![(s_0)_{init_state}]\!] =_{def} \{sys \in SYS \mid E = E_{sys} \wedge s_0 = init_state_{sys}\}.$$

□

In Beispiel 6.1 spezifizieren wir sowohl den Zustandsraum als auch den Initialzustand der Bibliothek unserer Beispielentwicklung.

3.4 Techniken der Datenspezifikation

Für die Repräsentation der in den Abschnitten 3.1 und 3.2 eingeführten Produktarten, existiert eine Vielzahl geeigneter Notationen und Techniken. Die folgenden Ansätze sind Beispiele für Notationen, die eine formale Semantik besitzen und auf Konzepten der algebraischen Spezifikation basieren:

- Objektorientierte, algebraische Spezifikationssprache OS aus [Bre91].

- SPECTRUM [BFG+93] und die darauf abgestützte Formalisierung der Entity-Relationship-Modellierung in [Het95].

Ohne formale Semantik, aber mit hohem Verbreitungsgrad, ist die

- UML (Unified Modeling Language), hier insbesondere Klassendiagramme, in Verbindung mit der OCL (Object Constraint Language) [OMG01].

Neben den algebraischen Techniken, die primär für die eigenschaftsorientierte Spezifikation von (abstrakten) Datentypen eingesetzt werden, können wir auch modellbasierte Techniken (Angabe eines expliziten Modells) anwenden, zum Beispiel zustandsbasierte Programmiersprachen.

Die modellbasierten Techniken eignen sich insbesondere für die „konstruktive" Definition konkreter Datenstrukturen und ihrer charakteristischen Operationen, so dass sie bevorzugt auf Implementierungsebene zum Einsatz kommen, beispielsweise um die (Entitäts-)Algebra *A* eines Systems (vergleiche Definition 2.6) *eindeutig* festzulegen, indem Sorten und Operationen etwa in Form einer objektorientierten Programmiersprache angegeben werden.

3.4.1 Algebraische Spezifikation

Wir halten den Ansatz der algebraischen Spezifikation zur Beschreibung der Entitätsrechenstrukturen der ganzheitlichen Perspektive für sehr geeignet, da die Grundidee dieses Ansatzes die implementierungsunabhängige Beschreibung von Datentypen ist. Datentypen werden durch Angabe ihrer Signatur und ihrer charakteristischen Eigenschaften spezifiziert. Die Eigenschaften werden mittels mehrsortiger, logischer Formalismen ausgedrückt. Üblicherweise ist es eine eingeschränkte Form von Logik erster Stufe, wie beispielsweise Gleichungslogik (siehe zum Beispiel [GTW78]) oder eine auf Horn-Klauseln basierende Gleichungslogik (siehe etwa [TWW82]). Es existieren aber auch Ansätze mit einer vollständigen Logik erster Stufe (mit Gleichheit), beispielsweise die Sprache SPECTRUM [BFG+93]. Ein umfassender Überblick über die algebraische Spezifikation wird in [Wir90] gegeben.

Die in den vorangehenden Abschnitten eingeführten Produktarten zur fachlichen Datenspezifikation sowie die zugehörigen Systemmodellelemente haben wir so gewählt, dass sie eine geeignete semantische Domäne für algebraische Spezifikationstechniken darstellen. *Signatur* und *Algebra* beziehungsweise *Rechenstruktur* sind hierbei die verwendeten Kernkonzepte. Zu beachten ist, dass wir, indem wir eine *Klasse* von Algebren und nicht eine einzelne Algebra als Element der Produktarten zur Datenspezifikation festgelegt haben, die Möglichkeit schaffen, algebraische Spezifikationen *lose* zu interpretieren und auf diese Weise das methodisch bedeutsame Prinzip der Unterspezifikation einsetzen zu können. In [Wir90] werden drei Semantiken algebraischer Spezifikationen unterschieden, wobei eine algebraische Spezifikation als ein Zwei-Tupel, bestehend aus einer Signatur Σ und einer Menge von Axiomen E, verstanden wird:

- Mit der *losen* Semantik wird einer Spezifikation die Menge *aller* Σ-Algebren (beziehungsweise Rechenstrukturen) zugeordnet, welche die Axiome aus E erfüllen.

- Die *initiale* Algebra Semantik wählt die (Isomorphieklasse der) *initialen* Algebren aus der Menge der die Axiome erfüllenden Σ-Algebren aus.
- Im Falle der *terminalen* Algebra Semantik wird die Isomorphieklasse der *terminalen* Algebren gewählt.

Welche Art von Semantik gewählt wird, ist in unserem Ansatz nicht Teil der Produktarten, sondern Teil der Spezifikationstechnik. Im Falle der terminalen und der initialen Semantik enthält die Menge C eines Produktes (Σ, ... , C) nur isomorphe Σ-Algebren. Sinnvoll wäre beispielsweise die Wahl der initialen Semantik, wenn die Erweiterbarkeit/Änderbarkeit (der Implementierung) einer Sorte eine wichtige „nicht-funktionale" Anforderung ist, da hier die Trägermengenelemente „möglichst viel Information" enthalten (denn nur die Elemente sind identisch, für die dies durch Axiome der Spezifikation gefordert wird).

Die praktische Relevanz algebraischer Spezifikationstechniken wird deutlich, wenn wir beispielsweise Gleichungsaxiome als Testfälle verstehen, was insbesondere dann nahe liegt, wenn für die Implementierung Techniken, wie zum Beispiel objektorientierte Programmiersprachen, eingesetzt werden, welche die Kapselung und das Verbergen von Zustandselementen/Variablen mittels Zugriffsoperationen unterstützen. In Abbildung 3.2 strukturieren wir die Spezifikation der Sorte *Medium* so, dass dieser Zusammenhang deutlich wird. Die Spezifikation ist wie folgt gegliedert:

- Liste der Konstruktor-Funktionen
- Liste der Selektor-Funktionen
- Beschreibung des „Effektes" (der Anwendung) einer Konstruktor-Funktion bezüglich der Selektor-Funktionen

3.4.2 Algebraische Spezifikation objektorientierter Klassen

Objektorientierte Modellierungstechniken sind weit verbreitet und auch für unseren Ansatz scheint die (syntaktische) Strukturierung von Sorten fachlicher Entitäten und ihren charakteristischen Operationen, in Form von *Klassen* im Sinne der Objektorientierung, sehr gut geeignet.

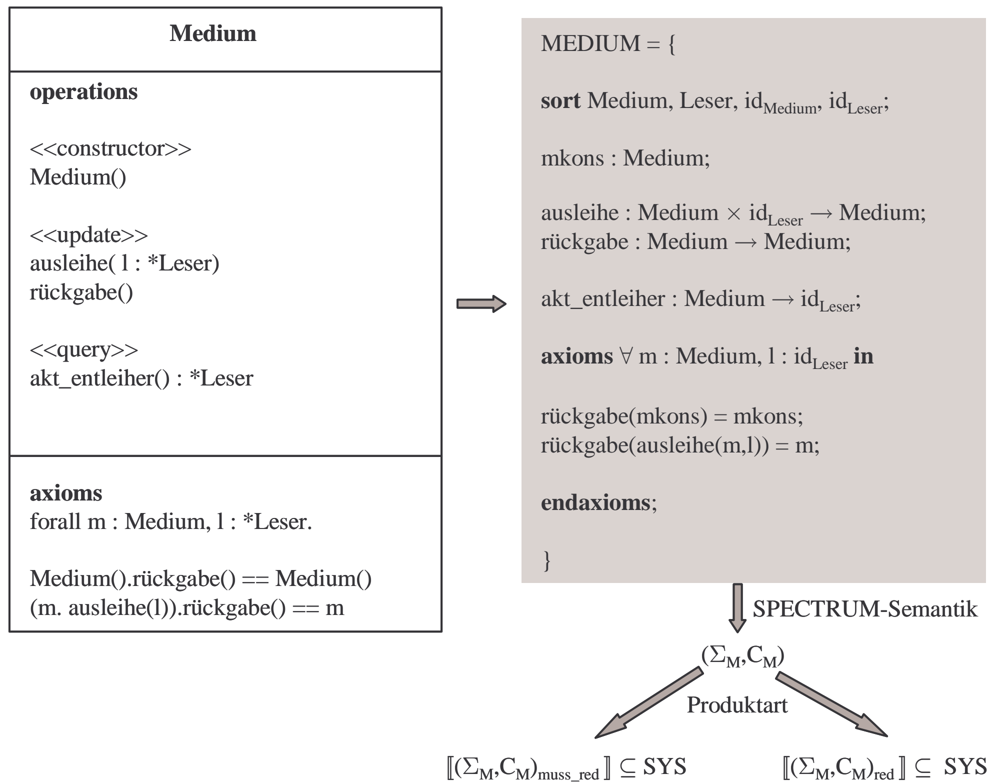


Abbildung 3.5:UML-basierte Klassenspezifikation und abgeleitete SPECTRUM-Spezifikation

In Abbildung 3.5 zeigen wir den Übergang von einer objektorientierten, algebraischen Spezifikation der Klasse *Medium*, zu einer algebraischen Spezifikation, die syntaktisch der Sprache SPECTRUM [BFG+93] entspricht, die das objektorientierte Konzept der *Klasse* nicht explizit enthält. Für die objektorientierte Spezifikation verwenden wir die Syntax der UML [OMG01].

Anhand des Übergangs von der UML-Spezifikation zur SPECTRUM-Spezifikation wird deutlich, dass mit der Deklaration einer objektorientierten Klasse implizit auch eine zugehörige Identifikatorsorte eingeführt wird (vergleiche beispielsweise [Rum96]). Die SPECTRUM-Spezifikation des Beispiels umfasst daher neben den Sorten *Medium* und *Leser* deren Identifikatorsorten id_{Medium} und id_{Leser}. Damit werden Attribute, deren Typ Referenzen auf Objekte sind, zum Beispiel

*akt_entleiher : *Leser*

(wobei **Leser* hier für den Typ der Referenzen auf Leser-Objekte steht) in Selektorfunktionen übersetzt, die ein Element einer Identifikatorsorte als Ergebnis liefern. Im Beispiel verdeutlicht dies die Funktion

$akt_entleiher : Medium \rightarrow id_{Leser}$.

Weiterhin wird durch den Übergang zwischen den Spezifikationsstilen der (implizite) Bezug von Objektmethoden auf ein Objekt dadurch explizit, dass die aus einer Objektmethode abgeleitete Funktion eine erweiterte Signatur aufweist. Im Beispiel wird etwa die Objektmethode

rückgabe()

abgebildet durch die Funktion

rückgabe : Medium → Medium.

wobei der Eingabeparameter den Ausgangszustand eines Objektes erfasst, und der Ergebnisparameter den Zustand(swert) angibt, den ein Objekt durch Anwendung der Objektmethode einnimmt. In unserem Systemmodell stellen wir den Bezug der zustandslosen Sicht der Entitätsrechenstruktur *A* eines Systems und der zustandsbezogenen Verhaltenssicht her, durch Ableitung der Zustandsalgebra *FS* aus Rechenstruktur *A* und Entitätsmenge *E* eines Systems (siehe hierzu Abschnitt 2.3.1)

In obigem Beispiel haben wir das objektorientierte Konzept der Subtypen nicht betrachtet. In [Bre91] wird eine objektorientierte, algebraische Spezifikationssprache angegeben, die auch dieses Konzept umfasst. Ebenso wie SPECTRUM hat auch diese Sprache eine lose Semantik. Als semantische Basis dienen hier partiell geordnete Signaturen und Algebren, um Subtyp-Beziehungen geeignet erfassen zu können. Ersetzen wir die „flachen“ Signaturen und Algebren durch partiell geordnete Signaturen und Algebren in unserem Systemmodell, so können wir Subtyping in unseren Ansatz integrieren.

3.4.3 Konstruktive Techniken

In den meisten Fällen wird die eigenschaftsorientierte, algebraische Spezifikation die geeignete Technik zur Charakterisierung fachlicher Daten/Entitäten darstellen. Grundsätzlich möglich, und für eine implementierungsnahe Sicht sinnvoll, ist auch die Verwendung konstruktiver, modellbasierter Techniken, durch die Trägermengen und Operationen eindeutig festgelegt werden, so dass die Menge *C* der Algebren einer Datenspezifikation einelementig und somit die Entitätsalgebra *A* eines Entwicklungssystems wegen

$C_f \subseteq C$ (was für alle Produktarten zur Datenspezifikation gilt) und $A \in C_f$ (vergleiche Definition 2.6)

eindeutig festgelegt ist.

Beispielsweise können wir die Entitätsrechenstruktur *A* mit Hilfe einer objektorientierten (zustandsbasierten) Programmiersprache, beispielsweise C++ oder Java, definieren. Mit der Definition einer Klasse legen wir eine entsprechende Wert- und Identifikatorsorte fest. Die Objektmethoden der Klasse entsprechen den charakteristischen Operationen der Sorte, deren Eigenschaften durch die Methodenrümpfe gegeben sind. Die Implementierung der Programmiersprache legt die Trägermengen und Funktionen der Algebra *A* fest.

Für die Definition konkreter Datentypen könnten, als mögliche Produktmodell-Erweiterung, spezielle Produktarten eingeführt werden, für die wir (bereits auf Ebene der abstrakten Syntax der Produkttypen) fordern:

$C \in Gen(\Sigma)$ (anstelle von $C \subseteq Gen(\Sigma)$, wie im Falle der oben eingeführten Produktarten zur Datenspezifikation).

In vielen Fällen wird, im Rahmen der (fachlichen) ganzheitlichen Perspektive, die Definition konkreter Datentypen nicht Teil einer Software-Entwicklung sein, da eine Konkretisierung nur für die durch das Softwaresystem abzubildenden Entitäten und deren Sorten relevant ist, so dass die Konkretisierung Bestandteil der komponentenbasierten Perspektive ist.

4 Prozessspezifikation

Als Teil der fachlichen Aufgabenstellung, die mit Hilfe eines zu entwickelnden Softwaresystems zu realisieren ist, sind die fachlichen Abläufe des Anwendungssystems zu charakterisieren. Im Fall von Informationssystemen sind dies die Abläufe von Unternehmen oder Organisationen im allgemeinen.

In unserem Systemmodell erfassen wir die fachlichen Abläufe eines Anwendungssystems durch die Prozessmenge P (vergleiche Definition 2.12). Die in diesem Kapitel behandelten Arten von Entwicklungsprodukten dienen der Spezifikation fachlicher Abläufe und somit der Spezifikation von Eigenschaften der Prozessmenge P.

Wie bereits in Abschnitt 2.3 erwähnt, sind Geschäftsprozess-Spezifikationen ein zentrales Element der Unternehmensmodellierung, so dass wir hierfür geeignete Produktarten benötigen. Im Wesentlichen gibt eine Geschäftsprozess-Spezifikation eine Menge alternativen Folgen fachlicher Aktivitäten, das heißt (fachlicher) Abläufe, an, welche der Erreichung eines Geschäftszieles dienen. Die Bearbeitung eines Vormerkungswunsches ist ein Beispiel für ein Geschäftsziel, das abhängig vom Zustand des vorzumerkenden Mediums und des Kontos des vorzumerkenden Lesers, entweder durch die Reservierung des Mediums für den jeweiligen Leser oder durch die Eintragung des Lesers am Ende der Liste der Vormerkungen, erreicht wird.

Für die Geschäftsprozess-Spezifikation unterscheiden wir drei Produktarten:

- Eine *exakte Spezifikation* gibt zu einem Geschäftsziel alle alternativen Abläufe, die zur Erreichung des Ziels als geeignet betrachtet werden, exakt an.
- Ein *Geschäftsprozess-Szenario (kann-Spezifikation)* gibt einige der alternativen Abläufe an, ohne Anspruch auf Vollständigkeit, während
- eine *muss-Spezifikation* eines Geschäftsprozesses einige der Eigenschaften angibt, die jeder der als zur Erreichung des Geschäftszieles geeignet betrachteten Abläufe zu erfüllen hat, ohne Anspruch auf Vollständigkeit bezüglich der geforderten Eigenschaften.

Die beiden letztgenannten Produktarten sind unter anderem für die Dokumentation der (Zwischen-)Ergebnisse von Bedeutung, die bei der schrittweisen Entwicklung einer exakten Geschäftsprozess-Spezifikation entstehen. Sind beispielsweise zu Beginn einer Entwicklung nur einige der geeigneten Alternativen bekannt, so kön-

nen diese durch ein Geschäftsprozess-Szenario festgehalten werden, zum Beispiel nur die fachlich wünschenswerten Abläufe (in denen keine technisch bedingten Fehler auftreten). Sind einige der Eigenschaften bekannt, die jede Alternative zu erfüllen hat, so können diese Eigenschaften durch muss-Spezifikationen dokumentiert werden. Ein Beispiel ist die Anforderung, dass, falls ein technischer Fehler bei der Umsetzung eines Geschäftszieles auftritt, eine geeignete Fehlerbehandlung stattzufinden hat. Mit Hilfe einer muss-Spezifikation kann diese Anforderung formuliert werden, ohne die „fehlerfreien" Abläufe zur Erreichung des betrachteten Geschäftszieles kennen zu müssen.

Damit schränken Szenarien die Menge der Alternativen, die in einer exakten Spezifikation festgelegt wird, „von unten" ein, während muss-Spezifikationen eine Einschränkung „von oben" darstellen. Seien X_s und X_m die in einem Szenario beziehungsweise einer muss-Spezifikation gegebene Menge von Alternativen, und X die exakte Menge, so muss sinnvollerweise gelten:

$X_s \subseteq X \subseteq X_m$.

In Abschnitt 4.1 zeigen wir anhand der in [WFMC99] gegebenen, informellen und praxisrelevanten Definition des Geschäftsprozess-Begriffes, dass eine Geschäftsprozess-Spezifiktion im Kern der Angabe einer Lebendigkeitseigenschaft entspricht, genauer gesagt einer sogenannten *response* Eigenschaft, wie sie etwa aus dem Bereich der temporalen Logik [MP92] bekannt ist. Um die drei oben genannten Arten von Geschäftsprozess-Spezifikationen im Systemmodell abbilden zu können, haben wir die Reaktionsabbildung *reak* in Definition 2.13 eingeführt, durch welche *response* Eigenschaften der Prozessmenge *P* induziert werden (vergleiche Definition 2.14).

Eine weitere Produktart führen wir ein, um zwischen (aus fachlicher Sicht) „erwünschten" und „unerwünschten" Alternativen zur Erreichung eines Geschäftszieles unterscheiden zu können. Beispielsweise muss gegebenenfalls eine aufgrund eines technischen Fehlers fehlgeschlagene Bearbeitung eines Vormerkungswunsches zwar als eine mögliche Alternative akzeptiert werden, falls technische Fehler unvermeidbar sind, jedoch ist dies keine „wünschenswerte" Alternative. Wir modellieren diese Unterscheidung, indem wir wünschenswerte Alternativen in Form von Fairness-Eigenschaften (der Prozesse aus P) formulieren.

Wie etwa aus [AS85] bekannt, kann jede Verhaltenseigenschaft eines verteilten Systems durch Kombination einer Lebendigkeits- und einer Sicherheitseigenschaft formuliert werden. Oben genannte Produktarten dienen der Spezifikation von (bestimmten Arten von) Lebendigkeitseigenschaften. In Abschnitt 4.2 führen wir die Produktart der Invarianten-Spezifikation ein, um Sicherheitseigenschaften der Prozessmenge P, das heißt der fachlichen, prozessorientierten Verhaltenssicht des Anwendungssystems, anzugeben. In [Web91] wird gezeigt, dass in Form von Invarianten, jede beliebige Sicherheitseigenschaft angegeben werden kann, so dass wir mit den bisher genannten Produktarten beliebige Verhaltenseigenschaften formulieren können.

In [MP92] wird eine sehr differenzierte Klassifikation von Sicherheits- und Lebendigkeitseigenschaften vorgeschlagen. Jede dieser Eigenschaftsklassen ist durch eine Normalform in Form eines temporallogischen Formelschemas definiert. Um diese Normalformen als Produktarten interpretieren zu können, übertragen wir die

temporallogischen Operatoren, mit Hilfe derer die Formelschemata aufgebaut sind, von dem in [MP92] verwendeten, und für die Temporallogik üblichen Modell der Spuren, auf unseren Prozessbegriff (aus Definition 8.9). Damit schaffen wir in Abschnitt 4.3 die Grundlage für die Erweiterung unseres Produktmodells um weitere Produktarten, die für die Verhaltensspezifikation verteilter Systeme geeignet sind.

Abbildung 4.1 gibt einen Überblick über die in diesem Kapitel eingeführten Arten von Entwicklungsprodukten.

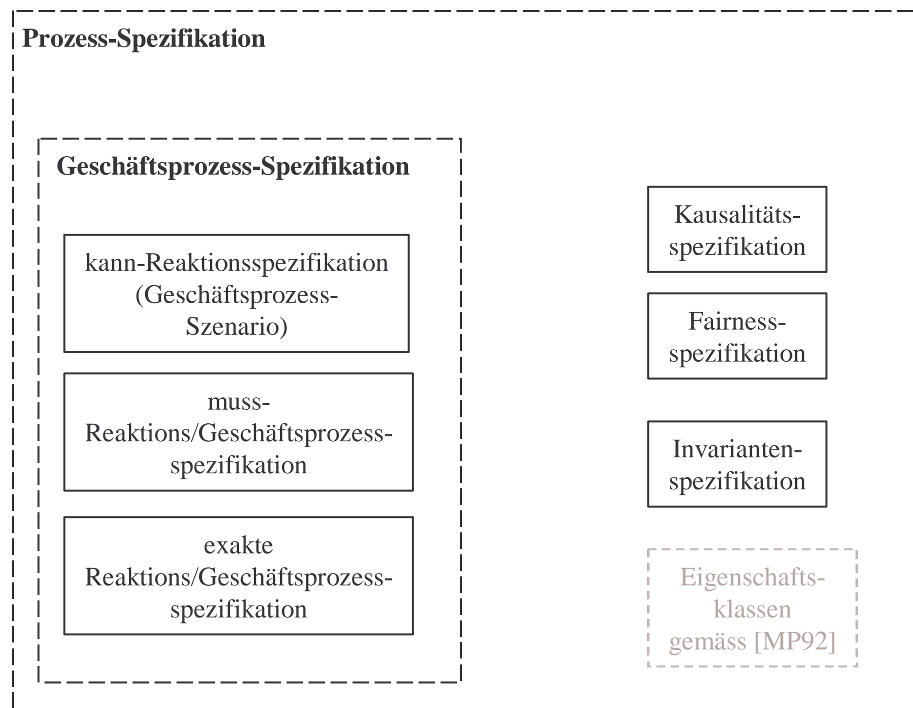


Abbildung 4.1: Produktarten zur Spezifikation von Prozesseigenschaften

4.1 Geschäftsprozess-Spezifikation

Die Spezifikation von Geschäftsprozessen ist, wie in Abschnitt 2.3 erwähnt, ein zentrales Element der Unternehmensmodellierung und damit der Erfassung der fachlichen Aufgabenstellung von Informationssystemen.

Aus Sicht der Softwareentwicklung können wir ein Unternehmen oder eine Organisation als ein verteiltes System verstehen, dass aus mindestens zwei verteilten Komponenten aufgebaut ist, dem Softwaresystem und dessen Umgebung, wobei die Komponenten durch Kooperation (unter anderem) Geschäftsprozesse realisieren.

Da eine Geschäftsprozess-Spezifikation im allgemeinen eine Menge von Abläufen (zur Erreichung eines Geschäftszieles) angibt, können wir eine Geschäftsprozess-Spezifikation als Spezifikation einer Verhaltenseigenschaft eines verteilten Systems auffassen. Es stellt sich die Frage, welcher Art von Verhaltenseigenschaft ein Geschäftsprozess entspricht ?

Wie aus [AS85] bekannt, stellen die sogenannten Sicherheits- und Lebendigkeitseigenschaften zwei grundlegende Klassen von Verhaltenseigenschaften dar. Die beiden Klassen werden informell wie folgt charakterisiert:

- *„Informally, a safety property stipulates that some `bad thing´ does not happen during execution."* [AS85, Lam77]
- *"Informally, a liveness property stipulates that a `good thing´ happens during execution."* [AS85, Lam77]

Eine weitergehende Differenzierung von Verhaltenseigenschaften finden wir etwa in [MP92], wo unter anderem die Klasse der sogenannten *response properties* als eine spezielle Form von Lebendigkeitseigenschaft (mit Hilfe temporallogischer Formeln) angegeben wird:

> *„A property that can be specified by a response formula is called a* response property. *[...] A normal form for response formulas is*
>
> $\Box(p \Rightarrow \Diamond q)$.
>
> *This formula states that every p-position is followed by or coincides with a q-position. Thus, we may interpret q as a guaranteed response to p."* [21]
>
> [MP92]

Wir nennen *response* Eigenschaften im Folgenden synonym auch *Reaktionseigenschaften.*

Um einen Bezug zu Geschäftsprozessen herzustellen, betrachten wir die in [WFMC99] gegebene Definition des Geschäftsprozess-Begriffs, da diese Definition

- zum einen zu allgemein akzeptierten Definitionen (vergleiche etwa [Dav93, Kos62, Pae98b, Schee98]) konsistent ist, und
- zum zweiten die Geschäftsprozess-Aspekte präzisiert, die für die Entwicklung von Softwaresystemen (welche Geschäftsprozesse realisieren) relevant sind.

Ein Geschäftsprozess wird in [WFMC99] als *Business Process* bezeichnet und wie folgt definiert:

---

[21] $\Box$ bezeichnet den *henceforth* Operator, für den gilt: „$\Box$p holds at a position j iff p holds at position j and all following positions – "from now on." " [MP92]

$\Diamond$ bezeichnet den *eventually* Operator, für den gilt: „$\Diamond$p holds at a position j iff p holds at some position k $\geq$j." [MP92]

> *"Business Process Definition: A set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships. [...]*
>
> *A business process has defined conditions triggering its initiation in each new instance (e.g. arrival of a claim) and defined outputs at its completion."*
>
> [WFMC99]

Erfassen wir die *defined triggering conditions* durch eine Formel *p*, und *the set of linked procedures* durch eine Formel *q*, dann lässt sich ein Geschäftsprozess durch die *response formula*

$$\Box(p \Rightarrow \Diamond q)$$

ausdrücken.

In unserem Systemmodell geben wir fachliches Verhalten durch die Prozessmenge

$$P \subseteq pcs_{fin}(EV, M)$$

an. In diesem Verhaltensmodell können wir die für einen Geschäftsprozess charakteristische auslösende Bedingung durch eine bestimmte Art von Ereignis abbilden, das heißt durch eine Ereignismarkierung $m \in M$. Die ausgelösten Reaktionen, die *linked procedures*, können wir durch eine Menge von (Isomorphieklassen von) Prozessen $Q \subseteq [pcs_{fin}(EV, M)]$ abbilden. Eine Geschäftsprozess-Spezifikation, mit Auslöser *m* und Reaktionsmenge *Q*, bedeutet damit folgende Anforderung an einen Prozess $p \in P$:

> Jedes Auftreten eines mit *m* markierten Ereignisses führt zu einem Auftreten eines Prozesses aus *Q*.

Um diese Art von Anforderungen an die Prozessmenge *P* eines Entwicklungssystems zu stellen, haben wir in Definition 2.13 die Abbildung

$$reak : M \rightarrow \wp([pcs_{fin}(EV, M)])$$

eingeführt. Falls für eine Markierung *m* und eine Prozessmenge *Q* gilt:

$$reak(m) = Q$$

so bedeutet dies (gemäss Definition 2.14), dass alle $p \in P$ die oben beschriebene Reaktionseigenschaft zu erfüllen haben. Somit legen die im Folgenden vorgestellten Produktarten zur Geschäftsprozess-Spezifikation Eigenschaften der Reaktionsabbildung *reak* fest.

Zu beachten ist, dass wir anstelle atomarer Reaktionen bestehend aus einem (!) Ereignis, wie in [MP92] und anderen temporallogischen Ansätzen (zum Beispiel [Lam94] und [Pra86]), Reaktionen in Form von Prozessen, und damit einer Menge von Ereignissen, angeben. Wie oben argumentiert, entspricht dies der Vorstellung von Geschäftsprozessen als *linked procedures* (Plural !) und führt, unserer Meinung nach, zu einer praktikablen und gut skalierenden Form der Spezifikation. Eine Beschränkung auf atomare Reaktionen bedeutet, nach unserer Ansicht, in vielen Fällen eine unnötig komplexe und damit schwer verständliche Darstellung der interessanten Verhaltenseigenschaften eines Systems.

Im allgemeinen existieren zu einem Auslöser mehrere Reaktionsalternativen. Auf jeden Fall notwendig ist eine Produktart zur exakten Festlegung der Menge der Reaktionsalternativen für einen gegebenen Auslöser. Wir nennen diese Produktart *exakte Reaktionsspezifikation* (oder synonym *exakte Geschäftsprozess-Spezifikation*). Eine solche exakte Spezifikation ist jedoch häufig das Ergebnis einer schrittweisen Entwicklung, innerhalb derer sich von zwei Seiten der exakten Lösung angenähert wird:

- Zum einen durch Angabe einiger (exemplarischer), aber nicht notwendigerweise aller Reaktionsalternativen. Das heißt, es wird eine Teilmenge der Alternativen festgelegt. Ein Beispiel wäre die Angabe der Reaktionen, innerhalb derer kein technischer Fehler auftritt.
- Zum zweiten durch Angabe von Eigenschaften, die jede Reaktionsalternative zu erfüllen hat, ohne damit notwendigerweise die Menge der Reaktionsalternativen bereits stark genug eingeschränkt zu haben, um die Menge der Alternativen exakt angegeben zu haben. Das heißt, es wird eine Obermenge der Alternativen angegeben. Ein Beispiel wäre Eigenschaft, dass jede Reaktion, innerhalb derer ein technischer Fehler auftritt, auch eine anschließende Fehlerbehandlung umfasst.

Um diese annähernde, schrittweise Entwicklung exakter Spezifikationen zu unterstützen, führen wir die Produktart der *kann-Reaktionsspezifikation* (synonym *Geschäftsprozess-Szenario*) ein, um eine Teilmenge von Alternativen anzugeben, sowie der *muss-Reaktionsspezifikation* (synonym *muss-Geschäftsprozess-Spezifikation*), um eine Obermenge anzugeben.

Schreiben wir

a führt zu X,

um auszudrücken, dass *X* eine Menge von Reaktionen auf einen Auslöser *a* darstellt, dann muss folgender Zusammenhang zwischen Reaktionsspezifikationen, die sich auf einen gemeinsamen Auslöser *a* beziehen, bestehen: Falls

- *a führt zu K* eine kann-Spezifikation,
- *a führt zu E* eine exakte Spezifikation und
- *a führt zu M* eine muss-Spezifikation

sind, dann muss gelten:

$K \subseteq E \subseteq M$.

In [Bro01] und [Kle97] wird ebenfalls zwischen diesen drei verschiedenen Sichten unterschieden.

In Definition 4.1 legen wir die drei genannten Produktarten zur Reaktionsspezifikation fest.

Definition 4.1 (Drei Arten von Reaktionsspezifikationen)

Sei M´ eine endliche Menge von Ereignismarkierungen und E´ eine abzählbare Menge von Ereignisbezeichnern. Eine Reaktionsspezifikation ist ein 2-Tupel

(m, X)

bestehend aus

- einer Ereignismarkierung $m \in M'$ und
- einer Menge von Isomorphieklassen von Prozessen $X \in \wp([pcs_{fin}(E', M')])$.

Wir unterscheiden drei Arten von Reaktionsspezifikationen

- *kann-Reaktionsspezifikation (m, X)$_{kann_reak}$*
- *muss-Reaktionsspezifikation (m, X)$_{muss_reak}$* und
- *exakte Reaktionsspezifikation (m, X)$_{reak}$*.

Die Semantik der drei Arten von Reaktionsspezifikationen definieren wir wie folgt:

- $[\![(m, X)_{kann_reak}]\!] =_{def} \{ sys \in SYS \mid m \in M_{sys} \wedge X \subseteq reak_{sys}(m) \}$
- $[\![(m, X)_{muss_reak}]\!] =_{def} \{ sys \in SYS \mid m \in M_{sys} \wedge reak_{sys}(m) \subseteq X \}$
- $[\![(m, X)_{reak}]\!] =_{def} \{ sys \in SYS \mid m \in M_{sys} \wedge X = reak_{sys}(m) \}$

□

Die unterschiedlichen Rollen, welche die verschiedenen Arten von Reaktionsspezifikationen im Rahmen einer Entwicklung spielen können, illustrieren die folgenden drei Beispiele, in denen wir die schrittweise Entwicklung einer exakten Spezifikation des Geschäftsprozesses zur Bearbeitung von Medien-Vormerkungen in einer Bibliothek betrachten. In Beispiel 4.1 gehen wir zunächst auf die Rolle von kann-Reaktionsspezifikationen ein.

Beispiel 4.1 (kann-Reaktionsspezifikation)

Wir betrachten, wie in den vorangegangenen Beispielen, die Entwicklung eines Bibliotheks-Softwaresystems. In diesem Beispiel gehen wir von folgendem, hypothetischen Stand eines Entwicklungsprojektes aus:

Es sei die (partielle Redukt-)Datenspezifikation MEDIUM´ aus Beispiel 3.2 gegeben, durch welche die Sorten für Medien- und Leser(konten)-Entitäten charakterisiert sind.

In einem nächsten Entwicklungsschritt sollen aus fachlicher Sicht, zum Beispiel durch eine Bibliothekarin, die Abläufe zur Bearbeitung eines Vormerkungswunsches spezifiziert werden. Softwaretechnische Aspekte, und damit beispielsweise durch technische Fehler bedingte Ablaufalternativen, sollen hierbei (noch) nicht berücksichtigt werden, so dass die zu erstellende Spezifikation keinen Anspruch auf Vollständigkeit bezüglich der möglichen Bearbeitungsalternativen von Vormerkungswünschen erhebt. Das bedeutet, dass die zu erarbeitende Spezifikation den Zweck einer kann-Reaktionsspezifikation, das heißt eines Geschäftsprozess-Szenarios, hat.

Um die Reaktionsspezifikation formulieren zu können, werden Ereignismarkierungen eingeführt, um die Äußerung eines Vormerkungswunsches, die Aufforderung (eines Lesers) zur Rückgabe eines entliehenen Mediums sowie die Bearbeitung einer Medien-Zuteilung an einen Leser, als Prozessereignis modellieren zu können. Da mit allen drei genannten Ereignisarten kein Zugriff auf Entitäten assoziiert wird, werden sie durch Aktionsbezeichner (vergleiche gegebenenfalls Definition 2.11) erfasst. Die Aktionsbezeichner *vormerkungswunsch*, *rückgabeaufforderung* und *zuteilungsbearbeitung* werden mit der in Abbildung 4.2 gegebenen Spezifikation, in Abhängigkeit von den Identifikatorsorten id_{Medium} und id_{Leser}, spezifiziert.

```
ACT = { enriches MEDIUM´;

-- Selektoren
eqstatus : Medium × status → Bool;

-- Generatoren von Aktionsbezeichnern
vormerkungswunsch :  idMedium × idLeser → Act;
rückgabeaufforderung :  idMedium → Act;
zuteilungsbearbeitung :  idMedium → Act;

axioms ∀ m : Medium, s : Status, l : idLeser  in

eqstatus(m,s) = (status(m) = s);

}
```

Abbildung 4.2: Spezifikation von Aktionsbezeichnern zur Medienvormerkung

Die Bearbeitung eines (konkreten) Vormerkungswunsches hängt davon ab, welches Medium für welchen Leser vorzumerken ist. Jedoch folgt die Bearbeitung immer demselben Schema, dass wir in Abbildung 4.3 in Form eines Aktivitätsdiagramms der UML [OMG01] angeben. Daraus geht hervor, dass der Zustand des vorzumerkenden Mediums einen Einfluss auf die Vormerkungsbearbeitung hat. Ist das Medium entliehen, wird der aktuelle Entleiher zur (rechtzeitigen) Rückgabe aufgefordert. Falls das Medium verfügbar ist, wird es dem nächsten vorgemerkten Leser zugeteilt.

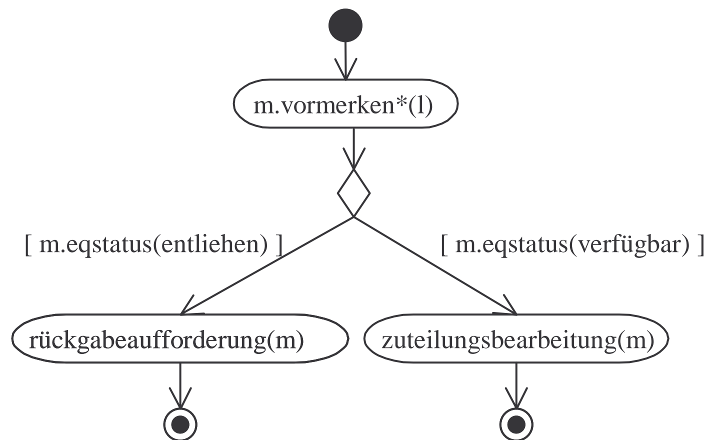


Abbildung 4.3: Prozessschema für die Bearbeitung eines Vormerkungswunsches

Die Beschreibung der Rückgabeaufforderung beziehungsweise der Zuteilungsbearbeitung durch einen Aktionsbezeichner ist (noch) sehr abstrakt. Mit dem Aktionsbezeichner *rückgabeaufforderung(m)* wird beispielsweise nicht die oben informell gegebene Aussage erfasst, dass der *aktuelle Entleiher* eines Mediums zur Rückgabe aufgefordert wird und *wie* diese Aufforderung auszuführen ist. Eine konkretere Beschreibung kann auf Prozessebene durch Verfeinerung, das heißt durch Ersetzen eines Ereignisses durch einen (Teil-)Prozess (vergleiche [GG98]), erreicht werden. Darüber hinaus kann eine Konkretisierung mit dem Wechsel in die komponentenbasierte Perspektive vorgenommen werden, indem den Aktionsbezeichnern bestimmte Interaktionen (mittels der Funktion *hi* aus Definition 2.28) zugeordnet werden.

Die Spezifikation von Aktionsbezeichnern aus Abbildung 4.2 macht deutlich, dass die Menge der Vormerkungswünsche (repräsentiert durch Aktionsbezeichner *vormerkungswunsch(m,l)*), die in einem System möglich sind, abhängig ist von der Menge der Medien und Leser des Systems. Analog dazu, sind auch die Reaktionen auf Vormerkungswünsche parametrisiert. Daher besteht eine enge Abhängigkeit zwischen der Menge der Aktionsbezeichner für Vormerkungswünsche und der Spezifikation der Systemreaktionen auf Vormerkungswünsche. Aus diesem Grund definieren wir ein Produkt von der Art eines Entwicklungsprädikates gemäss Definition 2.5, das sich zusammensetzt aus

- einer fachlichen Reduktspezifikation, mit der die notwendigen Aktionsbezeichner eingeführt werden, und
- einer Menge von kann-Reaktionsspezifikationen, die Verhaltenseigenschaften bezüglich aller möglichen Vormerkungswünsche umfasst.

Wir nennen dieses komponierte Entwicklungsprodukt VMR (Vormerkungsreaktion) und definieren es wie folgt:

$$\mathrm{VMR} =_{\mathrm{def}} (\, p_{\mathrm{VMR}} \,)_{\mathrm{pred}}$$

wobei

$$p_{VMR}(sys) \Leftrightarrow_{def} sys \in \left[\!\left[(\{Medium, Leser, Status\} \Sigma_{ACT}, C_{ACT})_{muss_red} \right]\!\right] \wedge$$
$$\forall m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}} .$$
$$sys \in \left[\!\left[\left(vormerkungswunsch^{A_{sys}}(m,l), vmreak(m,l) \right)_{kann_reak} \right]\!\right]$$

für alle $sys \in SYS$ gelte.

Dabei seien die Signatur Σ_{ACT} und die Klasse C_{ACT} von Σ_{ACT}-Algebren durch die, in Abbildung 4.2 gegebene, Spezifikation ACT charakterisiert.

Zudem sei die Funktion

$$vmreak : (A_{sys})_{id_{Medium}} \times (A_{sys})_{id_{Leser}} \to \wp([pcs_{fin}(EV_{sys}, M_{sys})])$$

für alle $m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}}$ definiert durch

$$vmreak(m,l) =_{def} \{[(E_x, \leq_x, \alpha_x)], [(E_y, \leq_y, \alpha_y)]\}$$

mit

$E_x = \{e_{x1}, e_{x2}, e_{x3}\}$,
$e_{x1} \leq_x e_{x2} \leq_x e_{x3}$,
$\alpha_x(e_{x1}) = (m.vormerkung\,*)^{FS_{sys}}(l)$,
$\alpha_x(e_{x2}) = (m.eqstatus)^{FS_{sys}}(entliehen^{FS_{sys}})$,
$\alpha_x(e_{x3}) = rückgabeaufforderung^{A_{sys}}(m)$

und

$E_y = \{e_{y1}, e_{y2}, e_{y3}\}$,
$e_{y1} \leq_y e_{y2} \leq_y e_{y3}$,
$\alpha_y(e_{y1}) = (m.vormerkung\,*)^{FS_{sys}}(l)$,
$\alpha_y(e_{y2}) = (m.eqstatus)^{FS_{sys}}(verfügbar^{FS_{sys}})$,
$\alpha_x(e_{x3}) = zuteilungsbearbeitung^{A_{sys}}(m)$

Die Prozesse, die wir mittels *vm_reak* einem Medienidentifikator *m* und einem Leseridentifikator *l* zuordnen, folgen dem in Abbildung 4.3 gegebenen Schema.

Interessant an dem Zusammenhang zwischen Prozessen und Aktivitätsdiagrammen ist, dass wir die Wächter/Guards eines Aktivitätsdiagramms (*m.eqstatus(*)* in Abbildung 4.3) als, mit Zustandsprädikaten markierte, Ereignisse interpretieren. *Fork-Konnektoren,* graphisch dargestellt durch $\diamond$, zeichnen Alternativen aus, so dass ein Aktivitätsdiagramm nicht auf einen einzelnen Prozess, sondern auf eine *Menge* von Prozessen abzubilden ist, in diesem Beispiel die zweielementige Menge $\{[(E_x, \leq_x, \alpha_x)], [(E_y, \leq_y, \alpha_y)]\}$.

Strukturell sehr große Ähnlichkeit zu Aktivitätsdiagrammen, mit dem Fork-Element zur Beschreibung von Alternativen, haben Ereignisstrukturen (*event structures*) [GG98, Win87b], die, im Unterschied zu unserem Prozessbegriff, zusätzlich

eine Konfliktrelation (zur Auszeichnung von Alternativen) umfassen. Der Übergang von Ereignisstrukturen zu unseren Prozessen ergibt sich direkt, durch Betrachtung der konfliktfreien „Ausführungen“ einer Ereignisstruktur (vergleiche hierzu gegebenenfalls [GG98]). Diese Ähnlichkeit macht die Eignung unseres Prozess- und damit Systemmodells, für die Anwendung verbreiteter Techniken der Ablaufspezifikation, deutlich. Neben Aktivitätsdiagrammen sind in diesem Zusammenhang beispielweise ereignisgesteuerte Prozessketten (EPKs) [Schee98], die für die Geschäftsprozessmodellierung eingesetzt werden, zu nennen.

□

Aufbauend auf Beispiel 4.1 zeigt das folgende Beispiele die Rolle von muss-Reaktionsspezifikationen.

Beispiel 4.2 (muss-Reaktionsspezifikation)

Das in Beispiel 4.1 gegebene Geschäftsprozess-Szenario (kann-Reaktionsspezifikation) berücksichtigt nur rein fachliche Reaktionen auf die Äußerung eines Vormerkungswunsches. Reaktionen, die sich aufgrund der Beteiligung eines Softwaresystems an der Vormerkungsbearbeitung und damit der Möglichkeit technischer Fehler ergeben können, sind noch nicht berücksichtigt.

```
ACT´ = { enriches ACT;

- Generatoren von Aktionsbezeichnern
techfehler : → Act;
fehlermeldung : → Act;

}
```

Abbildung 4.4:Spezifikation von Aktionsbezeichnern bezüglich technischer Fehler.

In unserem hypothetischen Entwicklungsprojekt soll in einem nächsten Entwicklungsschritt die Vormerkungsbearbeitung im Falle des Auftretens technischer Fehler spezifiziert werden. Auf technische Fehler soll mit einer Fehlermeldung reagiert werden.

Hierzu sind zunächst Markierungen einzuführen, durch welche das Ereignis des Auftretens eines technischen Fehlers sowie die Ausgabe einer Fehlermeldung abgebildet werden, konkret sind dies die Aktionsbezeichner *techfehler* und *fehlermeldung* (siehe Datenspezifikation ACT´ in Abbildung 4.4, die eine Erweiterung der Spezifikation ACT aus Abbildung 4.2 ist). Damit können die Reaktionsalternativen aus Beispiel 4.1 um die Behandlung technischer Fehler erweitert werden (siehe Aktivitätsdiagramm in Abbildung 4.5; die mit *fachliche_reaktion* markierte Box steht als syntaktische Abkürzung für das Diagramm aus Abbildung 4.3).

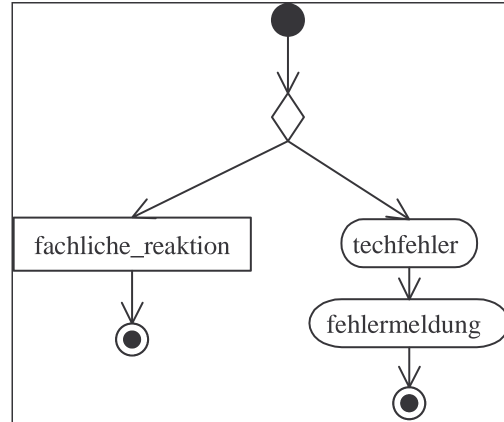


Abbildung 4.5: Erweitertes Prozessschema für die Bearbeitung eines Vormerkungswunsches, in dem technische Fehler berücksichtigt werden.

Analog zu Beispiel 4.1 definieren wir ein komponiertes Produkt:

$$\text{VMR}' =_{\text{def}} (\text{VMR}')_{\text{pred}}$$

mit

$$\begin{aligned} p_{\text{VMR}'}(\text{sys}) \Leftrightarrow_{\text{def}} \; & \text{sys} \in [\![(\Sigma_{\text{ACT}'}, C_{\text{ACT}'})_{\text{muss_red}}]\!] \wedge \\ & \forall m \in (A_{\text{sys}})_{\text{id}_{\text{Medium}}}, l \in (A_{\text{sys}})_{\text{id}_{\text{Leser}}} . \\ & \text{sys} \in [\![\left(\text{vormerkungswunsch}^{A_{\text{sys}}}(m,l), \text{vmreak}'(m,l) \right)_{\text{muss_reak}}]\!] \end{aligned}$$

für alle $\text{sys} \in \text{SYS}$. Dabei seien die Signatur $\Sigma_{\text{ACT}'}$ und die Klasse $C_{\text{ACT}'}$ von $\Sigma_{\text{ACT}'}$-Algebren durch die, in Abbildung 4.4 gegebene, Spezifikation ACT´ charakterisiert.

Zudem sei die Funktion

$$\text{vmreak}' : (A_{\text{sys}})_{\text{id}_{\text{Medium}}} \times (A_{\text{sys}})_{\text{id}_{\text{Leser}}} \rightarrow \wp([\text{pcs}_{\text{fin}}(\text{EV}_{\text{sys}}, M_{\text{sys}})])$$

für alle $m \in (A_{\text{sys}})_{\text{id}_{\text{Medium}}}, l \in (A_{\text{sys}})_{\text{id}_{\text{Leser}}}$ definiert durch

$$\text{vmreak}'(m,l) =_{\text{def}} \text{vmreak}(m,l) \cup \{[(E, \leq, \alpha)]\}$$

(mit *vmreak* aus Beispiel 4.1) mit

$$E = \{e_1, e_2\},$$
$$e_1 \leq e_2,$$
$$\alpha(e_1) = \text{techfehler}^{A_{\text{sys}}},$$
$$\alpha(e_2) = \text{fehlermeldung}^{A_{\text{sys}}}$$

Die Verwendung von muss-Spezifikationen in VMR´ ist motiviert durch folgende (hypothetische) Projektsituation:

> Die durch vm_reak´ beschriebenen Reaktionsmöglichkeiten umfassen die fachlich korrekten Reaktionen vollständig. Falls ein technischer Fehler auftritt, soll immer mit einer Fehlermeldung reagiert werden. Es wird davon ausgegangen, dass mit dem noch zu erstellenden Grobentwurf des Softwaresystems, weitere Anforderungen im Zusammenhang mit der Fehlerbehandlung zu berücksichtigen sein werden.

Das bedeutet, dass die in VMR´ erfasste Fehlervariante noch zu allgemein ist, um mit vm_reak´ eine exakte Reaktionsspezifikation zu formulieren.

□

Um die Bedeutung exakter Reaktionsspezifikationen zu illustrieren, setzen wir im folgenden Beispiel den hypothetischen Entwicklungsprozess der beiden vorangegangenen Beispiele fort.

Beispiel 4.3 (exakte Reaktionsspezifikation)

In diesem Beispiel gehen wir davon aus, dass das Produkt VMR´ aus Beispiel Beispiel 4.2 vorliegt. Weiterhin nehmen wir an, dass die Analyse der durch VMR´ gegebenen Spezifikation ergibt, dass eine Fehlermeldung als Reaktion auf einen technischen Fehler nicht ausreicht. Vielmehr sind zusätzlich die, Zusätzlich sind, durch die gegebenenfalls teilweise Bearbeitung eines Vormerkungswunsches entstandenen Effekte zu kompensieren (Recovery). Daher wird zunächst eine entsprechende Kompensationsaktion (in Form eines Aktionsbezeichners) definiert (siehe Abbildung 4.6).

ACT´´ = { **enriches** ACT´;

- Generatoren von Aktionsbezeichnern

kompensation : → Act;

}

Abbildung 4.6: Aktionsbezeichner für Fehlerkompensation eingeführt.

Zudem ist die muss-Spezifikation aus Beispiel 4.2 entsprechend zu verstärken, so dass sich das Prozessschema in Abbildung 4.7 ergibt.

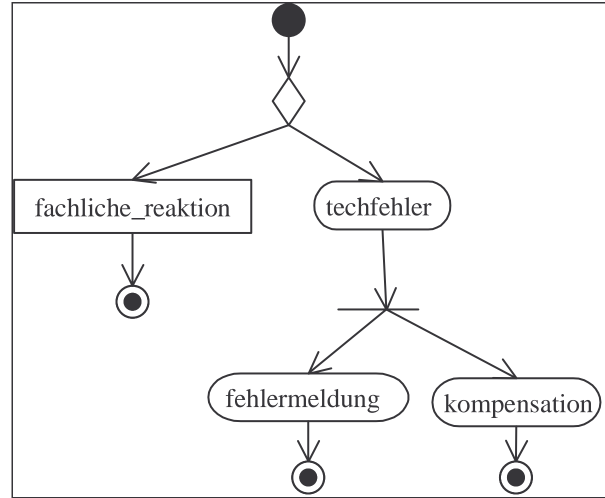


Abbildung 4.7: Vormerkungsbearbeitung um Fehlerkompensation erweitert.

Nachdem mit dieser Verstärkung alle zuständigen Projektbeteiligten darin übereinstimmen, dass damit alle Anforderungen an die Bearbeitung eines Vormerkungswunsches formuliert sind, werden in Produkt VMR´´ exakte Reduktspezifikationen verwendet:

$$VMR'' =_{def} (p_{VMR''})_{pred}$$

mit

$$\begin{aligned} p_{VMR''}(sys) \Leftrightarrow_{def}\ & sys \in [\![(\Sigma_{ACT''}, C_{ACT''})_{muss_red}]\!] \wedge \\ & \forall m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}} . \\ & sys \in [\![\left(vormerkungswunsch^{A_{sys}}(m,l), vmreak''(m,l) \right)_{reak}]\!] \end{aligned}$$

für alle $sys \in SYS$. Dabei seien die Signatur $\Sigma_{ACT''}$ und die Klasse $C_{ACT''}$ von $\Sigma_{ACT''}$-Algebren durch die, in Abbildung 4.6 gegebene, Spezifikation ACT´´ charakterisiert.

Zudem sei die Funktion

$$vmreak'' : (A_{sys})_{id_{Medium}} \times (A_{sys})_{id_{Leser}} \rightarrow \wp([pcs_{fin}(EV_{sys}, M_{sys})])$$

für alle $m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}}$ definiert durch

$$vmreak''(m,l) =_{def} vmreak(m,l) \cup \{[(E, \leq, \alpha)]\}$$

(bezüglich *vmreak* siehe Beispiel 4.1) mit

$E = \{e_1, e_2, e_3\}$,
$e_1 \leq e_2, e_1 \leq e_3$
$\alpha(e_1) = \text{techfehler}^{A_{sys}}$,
$\alpha(e_2) = \text{fehlermeldung}^{A_{sys}}$,
$\alpha(e_3) = \text{kompensation}^{A_{sys}}$

Zu beachten ist, dass die Hinzunahme des Kompensationsereignisses tatsächlich eine Verstärkung der Anforderungen und nicht eine beliebige Änderung der Anforderungen ist. Dies hängt mit der losen Interpretation von Reaktionsspezifikationen zusammen (vergleiche Definition 2.13 und Definition 2.14), durch die wir die explizit festgelegte Reaktion als *Teil*prozess des Gesamtverhaltens des Systems interpretieren.

□

4.1.1 Kausalität und Fairness

Mit den oben eingeführten Reaktionsspezifikationen beschreiben wir folgende Form von Anforderungen an das Systemverhalten:

> *Das Auftreten einer bestimmten Art von Ereignis hat zur Folge, dass irgendwann später eine bestimmte Art von (Teil-)Ablauf folgt.*

Die *Art* des auslösenden Ereignisses erfassen wir durch eine Ereignismarkierung. Die *Art* der Abläufe, die wir als Reaktion auf ein auslösendes Ereignis verstehen, erfassen wir durch eine Menge von Prozess-Isomorphieklassen.

Reaktionsspezifikationen lassen offen,

- in welchem zahlenmäßigen Verhältnis auslösende Ereignisse und Reaktionen stehen. Zum Beispiel ist eine Reaktionsspezifikation (auch dann) erfüllt, wenn auf einen Auslöser mehrere Reaktionen folgen, und umgekehrt
- mögliche Anforderungen an die Auswahl von bestimmten Reaktionsformen aus der Menge der möglichen Reaktionsformen.

Unter den erstgenannten Punkt ist die häufige Forderung einzuordnen, dass die Anzahl der Reaktionen mit der Anzahl der Auslöser überein zu stimmen hat. Wir sprechen davon, dass Auslöser und Reaktion *strikt kausal* zueinander sind. Anforderungen dieser Art erfassen wir durch die Produktart der sogenannten *Kausalitätsspezifikationen*. Für die Definition dieser Produktart benötigen wir folgende Hilfsfunktion:

Funktion #*

$$\#^* : \wp([\text{pcs}_{\text{fin}}(E', M')]) \times \text{pcs}_{\text{fin}}(E', M') \rightarrow \mathbb{N}$$

wobei für alle $X \in \wp([pcs_{fin}(E',M')]), p \in pcs_{fin}(E',M')$ gelte[22]:

$$\#^*(X,p) =_{def} |\{q \in pcs_{fin}(E',M') | q \prec p \wedge [q] \in X\}|$$

□

#*(X,p) liefert die Anzahl der Vorkommnisse von Prozessen aus X in p. Die Isomorphieklassen in X können wir als Prozessschemata verstehen, so dass #* die Anzahl von Instanzen dieser Schemata, die in p vorkommen, liefert. Hierbei ist zu beachten, dass, falls für ein q gilt, $\exists q'.\, q' \prec q \wedge [q'],[q] \in X$ (das heißt X umfasst auch einen Teilprozess eines Elementes q) ein Vorkommen von q in p im Ergebnis von #*(X,p) mehrfach eingeht, da eine Instanz von q eine Instanz von q´ umfasst.

Definition 4.2 (Kausalitätsspezifikation)

Sei M´ eine endliche Menge von Ereignismarkierungen und E´ eine abzählbare Menge von Ereignisbezeichnern. Eine *Kausalitätsspezifikation* ist ein 2-Tupel

(m, X)

bestehend aus

- einer Ereignismarkierung $m \in M'$ und
- einer Menge von Isomorphieklassen von Prozessen $X \in \wp([pcs_{fin}(E',M')])$.

Die Semantik einer Kausalitätsspezifikation definieren wir wie folgt[23]:

$$[\![(m,X)_{kausal}]\!] =_{def} \{sys \in SYS | \forall p \in P_{sys}.(\#(m©p) = \#^*(X,p))\}$$

□

Kommen wir zu dem zweiten angesprochenen Punkt, den Reaktionsspezifikationen offen lassen. Einschränkungen bezüglich der Auswahl von Reaktionsformen ergeben sich aus der Klassifikation von Reaktionsformen in (fachlich) korrekte und fehlerbehaftete Reaktionen. Üblicherweise werden wir von einer Realisierung fordern, dass sie nicht *nur* Fehlerfälle realisiert, beziehungsweise sogar, dass, wenn oft genug angefordert/ausgelöst, eine (fachlich) korrekte/erwünschte Reaktion eintritt. Anforderungen dieser Art sind unter dem Begriff der *starken Fairness* bekannt (siehe [Fran86]). Um die starke Fairness von Prozessen bezüglich einem Auslöser-Reaktionspaar zu formulieren, führen wir die folgende Produktart ein:

---

[22] mit $q \prec p$ steht für „q ist Teilprozess von p“ (vergleiche Definition 8.10).

[23] m©p steht für die Projektion auf mit m markierte Ereignisse in Prozess p (vergleiche Definition 8.11).

Definition 4.3 (Fairnessspezifikation)

Sei M´ eine endliche Menge von Ereignismarkierungen und E´ eine abzählbare Menge von Ereignisbezeichnern. Eine *Fairnessspezifikation* ist ein 2-Tupel

(m, X)

bestehend aus

- einer Ereignismarkierung $m \in M´$ und
- einer Menge von Isomorphieklassen von Prozessen $X \in \wp([\mathrm{pcs}_{fin}(E´, M´)])$.

Die Semantik einer Fairnessspezifikation definieren wir wie folgt:

$$[\![(m,X)_{sfair}]\!] =_{def} \{ sys \in SYS \mid \forall p \in P_{sys} . \\ \big((\#(m \copyright p) = \infty) \Rightarrow \big(\#^{*}(X,p) = \infty\big)\big)\}$$

□

Als Randbedingung für die sinnvolle Verknüpfung von Reaktionsspezifikationen mit Kausalitäts- und Fairnessspezifikationen ergibt sich für

$(m,Y)_{*_reak} \wedge (m,X)_{kausal}$

und

$(m,Y)_{*_reak} \wedge (m,X)_{sfair}$

dass $X \subseteq Y$ gelte.

Das folgende Beispiel zeigt eine sinnvolle Kombination von Reaktions- und Fairness-Spezifikation.

Beispiel 4.4 (Fairness-Spezifikation)

Die kann-Reaktionsspezifikation aus Beispiel 4.1 gibt die Reaktionen auf einen Vormerkungswunsch an, in denen keine technischen Fehler auftreten und die wir somit als die (aus fachlicher Sicht) „erwünschten“ Reaktionsalternativen verstehen können. Wir können dies durch folgende Fairness-Spezifikationen festlegen (wobei wir uns auf die Aktion *vormerkungswunsch* und die Funktion *vmreak* aus Beispiel 4.1 beziehen):

$$\left(\text{vormerkungswunsch}^{A_{sys}}(m,l), \text{vmreak}(m,l)\right)_{fair}$$

für alle $m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}}$ eines Systems $sys \in SYS$.

□

4.2 Sicherheitseigenschaften

Reaktionseigenschaften sind eine Form von Lebendigkeitseigenschaften. In [AS85] wird gezeigt, dass jede Verhaltenseigenschaft durch eine Kombination von Lebendigkeits- und Sicherheitseigenschaft ausgedrückt werden kann. Bevor wir zu einer Produktart für die Beschreibung von Sicherheitseigenschaften kommen, gehen wir kurz näher auf Lebendigkeits- und Sicherheitseigenschaften ein:

In [AS85] beziehen sich die Begriffe der Lebendigkeits- und Sicherheitseigenschaft auf ein Modell, in dem Systemverhalten durch Zustandsspuren modelliert wird. In [Web91] wurden diese Begriffe auf Aktionsspuren und Spurprädikate übertragen. Diese Formulierung lässt sich wiederum folgendermaßen direkt auf unsere ganzheitliche Perspektive, mit der Verhaltensmodellierung durch Prozesse, übertragen:

Definition 4.4 (Sicherheits- und Lebendigkeitsprädikate über Prozessen)

Sei E eine Menge von Ereignisbezeichnern und M eine Menge von Ereignismarkierungen. Ein Prädikat

$$S : pcs_{fin}(E,M) \rightarrow \mathbb{B}$$

ist ein *Sicherheitsprädikat*, falls für alle $p \in pcs_{fin}(E,M)$ gilt:

$$S(p) \Leftrightarrow \forall q \in pcs^{*}(E,M).q \sqsubseteq p \Rightarrow S(q).$$

Ein Prädikat

$$L : pcs_{fin}(E,M) \rightarrow \mathbb{B}$$

ist ein *Lebendigkeitsprädikat*, falls für alle $q \in pcs^{*}(E,M)$ gilt:

$$\exists p \in pcs_{fin}(E,M).q \sqsubseteq p \wedge L(p).$$

□

Von einem Sicherheitsprädikat wird damit gefordert, dass ein Ablauf genau dann sicher ist (das heißt er erfüllt ein Sicherheitsprädikat), wenn alle seine endlichen Abläufe sicher sind. Von einem Lebendigkeitsprädikat wird gefordert, dass jeder endliche Ablauf zu einem lebendigen Ablauf erweitert werden kann.

Nachdem wir mit den oben eingeführten Produktarten zur Reaktionsspezifikation eine Art von Lebendigkeitseigenschaften behandelt haben, kommen wir jetzt zur

Spezifikation von Sicherheitseigenschaften. Eine verbreitete Beschreibung von Sicherheitseigenschaften ist die sogenannte *Invariantenform*. Darunter wird ein Prädikat der Bauart

$$P(p) =_{def} \forall q \in pcs^{*}(E,M).q \sqsubseteq p \Rightarrow Q(q), \text{ für alle } p \in pcs_{fin}(E,M),$$

verstanden. Ein solches Prädikat besagt, dass jedes endliche Präfix von p die Eigenschaft Q zu erfüllen hat.

Zu beachten ist, dass jede Sicherheitseigenschaft in Invariantenform beschrieben werden kann (siehe hierzu [Web91]).

In [Bro89] werden zwei typische Arten von Sicherheitseigenschaften in Invariantenschreibweise angegeben (sei $p \in pcs_{fin}(E,M)$):

- ein mit b markiertes Ereignis kann höchstens gleich oft wie ein mit a markiertes Ereignis vorkommen:

  $$\forall q \in pcs^{*}(E,M).q \sqsubseteq p \Rightarrow \#(©(a,p)) \geq \#(©(b,p))$$

- zwischen mit a und b markierten Ereignissen besteht eine zyklische, kausale Abhängigkeit:

  $$\forall q \in pcs^{*}(E,M).q \sqsubseteq p \Rightarrow 0 \leq \#(©(a,p)) - \#(©(b,p)) \leq 1$$

Häufig stellen Verhaltenseigenschaften eine Kombination aus Invarianten- und Reaktionseigenschaften dar (vergleiche Beispiel 4.5). Daher führen wir die Produktart der *Invariantenspezifikation* (zur Formulierung von Sicherheitseigenschaften) ein.

Definition 4.5 (Invariantenspezifikation)

Sei E eine Menge von Ereignisbezeichnern und M eine Menge von Ereignismarkierungen. Eine Invariantenspezifikation besteht aus

- einem Prädikat $Q : pcs^{*}(E,M) \rightarrow \mathbb{B}$.

Die Semantik einer Invariantenspezifikation definieren wir wie folgt:

$$[\![(Q)_{inv}]\!] =_{def} \{ sys \in SYS \mid \forall p \in P_{sys}, q \in pcs^{*}(EV, M_{sys}). \; q \sqsubseteq p \Rightarrow Q(q) \}$$

□

Beispiel 4.5 illustriert den Einsatz einer Invariantenspezifikation.

Beispiel 4.5 (Invariantenspezifikation)

Durch das folgende Produkt *Zuteilungsauslöser (ZA)* legen wir fest, dass die Zuteilungsbearbeitung eines Mediums immer durch einen Vormerkungswunsch oder eine Rückgabe des Mediums ausgelöst wird. Damit schließen wir unter anderem aus, dass Zuteilungen „spontan“ stattfinden. Dies deutet darauf hin, dass die Zuteilungsbearbeitung nicht direkt an der Schnittstelle eines Bibliothekssystem angefor-

dert werden kann, sondern eine „interne“ Operation ist, die durch „Schnittstellen-operationen“ impliziert wird.[24]

$$ZA =_{def} (p_{ZA})_{pred}$$

mit

$$\begin{aligned} p_{ZA}(sys) \Leftrightarrow_{def}\ & sys \in [\![(\Sigma_{ACT}, C_{ACT})_{muss_red}]\!] \wedge \\ & \forall m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}} . \\ & sys \in [\![(Q(m,l))_{inv}]\!] \end{aligned}$$

für alle $sys \in SYS$, wobei

$$Q(m,l) : pcs_{fin}(EV, M_{sys}) \rightarrow \mathbb{B}$$

für alle $p \in pcs_{fin}(EV, M_{sys})$ wie folgt definiert sei:

$$\begin{aligned} Q(m,l)(p) \Leftrightarrow_{def}\ & \Big(\#\Big(©\Big(vormerkungswunsch^{A_{sys}}(m,l), p \Big)\Big) + \\ & \#\Big(©\Big(\big(m.rückgabe^{*}\big)^{FS_{sys}}(\ ), p \Big)\Big)\Big) \geq \\ & \#\Big(©\Big(zuteilungsbearbeitung^{A_{sys}}(m), p \Big)\Big) \end{aligned}$$

Die Signatur Σ_{ACT} und die Klasse C_{ACT} von Σ_{ACT}-Algebren seien durch die, in Beispiel 4.1 und Abbildung 4.2 gegebene, Spezifikation ACT charakterisiert.

□

Sicherheitseigenschaften, die (nur) *eine* Entität betreffen, zum Beispiel

> *„zu jeder erfolgreichen Zuteilung von m an l lässt sich* jeweils *eine voran gegangene erfolgreiche Vormerkung von l für m finden.“*

werden in unserem Ansatz sinnvollerweise nicht als Sicherheitseigenschaften formuliert, sondern durch Spezifikation der Entitätssorten, da wir aus diesen die zulässigen Zustandsübergänge ableiten (vergleiche Abschnitt 2.3.1).

Aus diesem Grund erscheint in unserem Ansatz der Einsatz von Invariantenspezifikationen in erster Linie für Sicherheitseigenschaften interessant, die Zusammenhänge zwischen

- Aktionen,
- Aktionen und Zustandsprädikaten oder Zustandsänderungen sowie
- Zustandsaspekten unterschiedlicher Entitäten

erfassen.

[24] In Beispiel 4.1 wurde die Zuteilungsbearbeitung als Bestandteil der Bearbeitung eines Vormerkungswunsches festgelegt.

4.3 Prozessspezifikation und temporale Logik

Die temporale Logik ist ein sehr gut verstandener, formaler Ansatz zur Spezifikation und Verifikation von Systemverhalten. Aus diesem Grund skizzieren wir den Zusammenhang unseres Ansatzes zur temporalen Logik.

Klassischerweise basiert die temporale Logik auf einem Systemmodell, in dem Verhalten durch Spuren von Systemzuständen modelliert wird (siehe zum Beispiel [Eme90] und [MP92]).

Temporallogische Operatoren, wie $\Diamond$ *(sometimes oder eventually)* und $\Box$ *(always oder henceforth)*, lassen sich leicht auf ein Kausalrelationsmodell übertragen (vergleiche zum Beispiel [Pra86] und [Bro89]). Für jedes Prädikat

$$Q : pcs_{fin}(E,M) \to \mathbb{B}$$

seien

$$\Box Q(p) : pcs_{fin}(E,M) \to \mathbb{B} \text{ und}$$

$$\Diamond Q(p) : pcs_{fin}(E,M) \to \mathbb{B}$$

mit

$$\Box Q(p) \Leftrightarrow_{def} (\forall q \in pcs_{fin}(E,M).q \sqsubseteq p \Rightarrow Q(p \setminus q))$$

beziehungsweise

$$\Diamond Q(p) \Leftrightarrow_{def} \exists q \in pcs_{fin}(E,M).q \sqsubseteq p \wedge Q(p \setminus q)$$

für alle $p \in pcs_{fin}(E,M)$.

Mit Hilfe dieser Operatoren können wir Verhaltenseigenschaften der ganzheitlichen Perspektive unseres Systemmodells formulieren.

In [MP92] werden Verhaltenseigenschaften auf Basis der temporalen Logik klassifiziert. Jede Eigenschaftsklasse ist durch eine temporallogische Normalform charakterisiert. Zum Beispiel ist die Klasse der sogenannten *response properties* charakterisiert durch

$$\Box(P \Rightarrow \Diamond Q)$$

wobei P und Q Prädikate sind, die sogenannten *past formulas* entsprechen. *Past formulas* sind Formeln, die keine *future operators (next, henceforth, eventually, unless)*, enthalten.

Folgende Eigenschaftsklassen werden in [MP92] unterschieden:

- *Safety*
- *Guarantee*
- *Obligation*
- *Response*
- *Persistence* und

- *Reactivity*.

Die Eigenschaftsklassen lassen sich in Forschritts- (*progress,* alle Klassen außer die Klasse *safety*) und Sicherheitseigenschaften (*safety*) partitionieren und hierarchisch ordnen. Hierarchie bedeutet in diesem Fall, dass eine hierarchisch niedrigere Eigenschaft ein Spezialfall einer hierarchisch höheren Eigenschaft ist (Inklusion).

> *„ [...] the safety class imposes a requirement, represented by a past formula, that must hold at all positions of a computation. The progress classes, on the other hand, specify a requirement that should eventually be fulfilled, and are therefore associated with progress toward fulfillment of the requirement. The progress classes differ from one another in the conditions and frequency at which the requirement is to be fulfilled."* [**MP92**]

In [MP92] werden den verschiedenen Eigenschaftsklassen entsprechend mächtige Sprachen und Automatenmodelle zugeordnet sowie geeignete Verifikationstechniken vorgestellt.

Die (Normalformen der) Eigenschaftsklassen können wir als Produktarten verstehen.

5 Komponentenbasierte Spezifikation

Die Spezifikation des zu entwickelnden Softwaresystems ist die zentrale Aufgabe einer Softwareentwicklung. In unserem Systemmodell erfassen wir das zu entwickelnde Softwaresystem und dessen Umgebung in Form eines Komponenten-Netzwerkes, dass komponiert ist aus dem Netzwerk, welches das Softwaresystem repräsentiert und dem Netzwerk, welches die Umgebung des Softwaresystems darstellt.

Im allgemeinen entstehen im Rahmen einer Softwareentwicklung mehrere Sichten zur Spezifikation der genannten Netzwerke. Diese spezifikatorischen Sichten, wie beispielsweise die konzeptuelle und die technische Architektur eines Systems aus [HNS99] oder die unterschiedlichen hierarchischen Ebenen der Erzeugnisstruktur des V-Modells [BDI97], sind häufig selbst wieder in Form von Komponenten-Netzwerken gegeben. In unserem Systemmodell erfassen wir die spezifikatorischen Sichten einer Entwicklung daher durch eine Menge von Netzwerken, die wir die *Spezifikationsnetzwerke* (eines Entwicklungssystems) nennen (vergleiche Abschnitt 2.5.4).

Um Spezifikationsnetzwerke zu charakterisieren, benötigen wir Produktarten zur Angabe von Netzwerk- und Komponenten-Eigenschaften. In den folgenden Abschnitten führen wir die grundlegenden Produktarten für diesen Zweck ein, die wiederum Bausteine für umfassende Produktarten, wie die sogenannte Schnittstellen- und Realisierungsarchitektur (vergleiche Abschnitt 6.4), darstellen.

In Abschnitt 5.1 gehen wir auf Produktarten zur Spezifikation der strukturellen Aspekte von Netzwerken ein. Strukturelle Aspekte, wie Kanalverbindungen zwischen Komponenten, sind Voraussetzung für Verhaltensaspekte von Komponenten und Netzwerken, da die Verhaltensmöglichkeiten durch die Struktur bestimmt sind.

Für die Darstellung des Zusammenwirkens der Komponenten eines Netzwerkes ist ein komponentenübergreifender Blickwinkel auf Verhaltensaspekte sinnvoll. Die entsprechenden Produktarten führen wir in Abschnitt 5.2 ein.

Das Verhalten eines Netzwerkes wird auf eindeutige Weise durch das Verhalten seiner Komponenten bestimmt. Für die Verhaltensspezifikation einzelner Komponenten geben wir die Produktart der Automatenspezifikation in Abschnitt 5.3 an. Aufgrund des operationellen Charakters eignen sich Automatenspezifikationen insbesondere für den Feinentwurf (und als Implementierungsvorlage) von Komponenten. Zudem stellen Automaten Komponentenzustand und Interaktionsverhalten in einen Zusammenhang. Dadurch sind sie ein wichtiges Mittel, um die Realisierung von zustandsbezogenem Prozessverhalten (der ganzheitlichen Perspektive)

durch das rein auf Interaktion beruhende, beobachtbare Komponentenverhalten (der Komponenten des Realisierungsnetzwerkes) anzugeben.

Ergänzend zu Automaten geben wir in Abschnitt 5.4 eine Produktart an, durch die (ausschließlich) das beobachtbare (Interaktions-)Verhalten einzelner Komponenten charakterisiert werden kann. Diese Form der Verhaltensspezifikation ist etwa für Komponenten geeignet, die keine fachlichen Daten realisieren und reine Steuerungsaufgaben übernehmen.

Wir schließen das Kapitel mit Abschnitt 5.5 ab, in dem wir die Spezifikation der Realisierungsbeziehung zwischen ganzheitlicher und komponentenbasierter Perspektive, das heißt zwischen fachlicher Aufgabenstellung und deren Realisierung durch Software- und Umgebungsnetzwerk, diskutieren. Hierbei führen wir unter anderem Produktarten zur Spezifikation der Realisierung von Prozessereignissen ein.

5.1 Statische Struktur

Ein grundlegender Aspekt von Komponenten und (Komponenten-)Netzwerken ist deren statische Struktur. Unter der Struktur einer Komponente verstehen wir dessen syntaktische Schnittstelle, das heißt die Menge der Ein- und Ausgabekanäle. Die Struktur eines Netzwerkes ist bestimmt durch die Menge der Netzwerkkomponenten und deren syntaktische Schnittstellen. Da wir, wie in Abschnitt 2.5 beschrieben, von einer *statischen* Komponenten- und Netzwerkstruktur ausgehen, das heißt die Komponentenmenge von Netzwerken sowie die syntaktische Schnittstelle jeder Komponente nehmen wir als invariant (für die gesamte Systemlebensdauer) an, sprechen wir von der statischen Struktur von Komponenten und Netzwerken.

Die statische Struktur ist ein grundlegender Aspekt, da sie einerseits unabhängig von Verhaltensaspekten festgelegt werden kann, andererseits aber Vorgaben für Verhalten macht. Letzteres hängt damit zusammen, dass in unserem Systemmodell Komponenten durch Kanalverbindungen zu Netzwerken komponiert werden. Die Kanalverbindungen bestimmen wiederum die Interaktionsmöglichkeiten und damit das Verhalten, da in unserem Modell Interaktion der Grundbaustein des beobachtbaren Komponentenverhaltens ist (vergleiche Abschnitte 2.5.2 und 8.4.1).

Häufig ist die Festlegung der statischen Struktur eines Netzwerkes ein erster Schritt im Entwurf einer Systemarchitektur, auf den Schritte zur Verhaltensspezifikation folgen. Geeignete Notationen sind Strukturdiagramme, wie zum Beispiel die sogenannten System-Struktur-Diagramme aus [SHB96] und die *actor*- und *binding*-Diagramme der Methode ROOM [SGW94]. Wir verwenden in dieser Arbeit Strukturdiagramme, in denen Komponenten als Kästen und Kanäle als Pfeile dargestellt werden. Abbildung 5.1 zeigt ein Strukturdiagramm, das drei Komponenten und drei Kanäle umfasst.

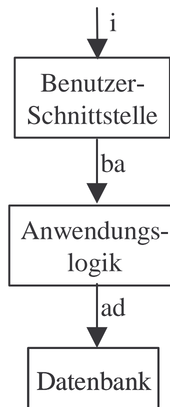


Abbildung 5.1: Statische Struktur Diagramm

Die im folgenden eingeführten Produktarten dienen der Charakterisierung der statischen Struktur einzelner Komponenten sowie von Spezifikationsnetzwerken eines Entwicklungssystems. Abbildung 5.2 gibt einen Überblick der Produktarten.

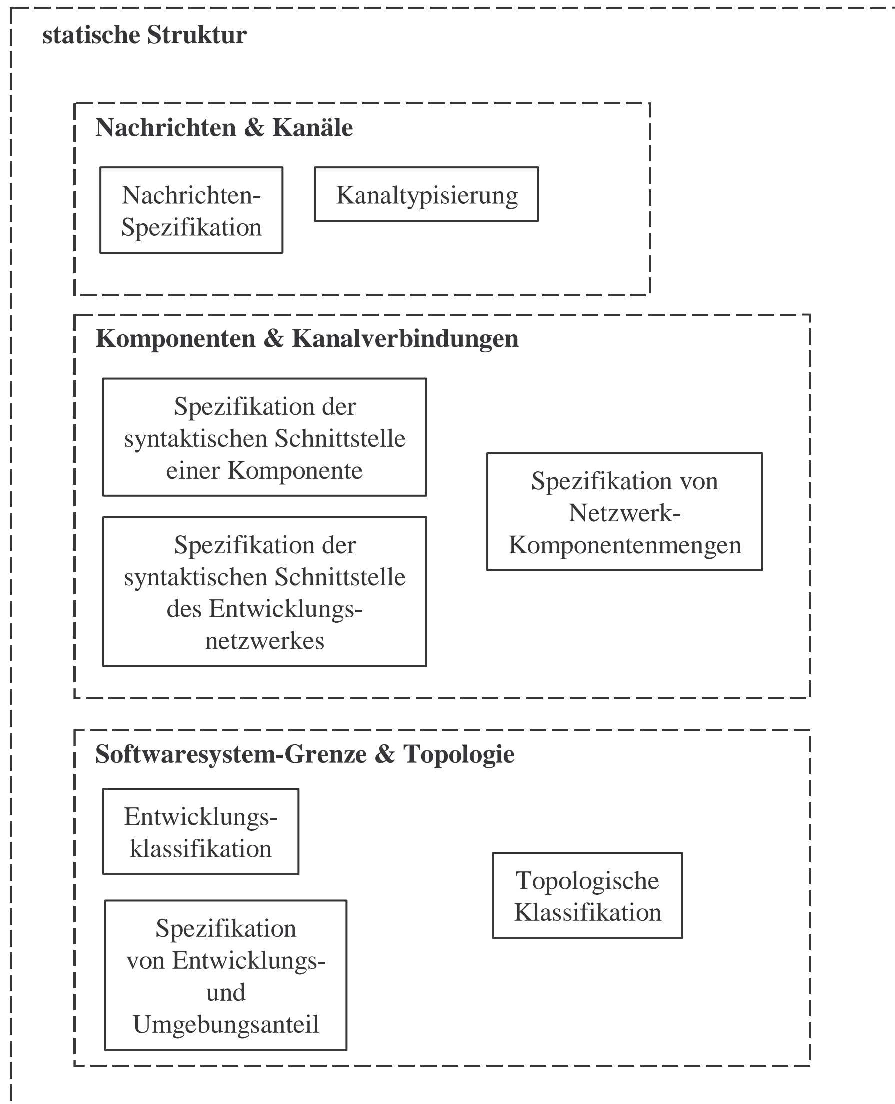


Abbildung 5.2: Produktarten zur Spezifikation von Strukturaspekten

5.1.1 Nachrichten und Kanäle

Elemente einer statischen Struktur sind Kanäle. Charakteristisch für einen Kanal ist die Menge der über ihn übertragbaren Nachrichten. Um einem Kanal dessen Nachrichtenmenge zuordnen zu können, müssen Nachrichten spezifiziert werden. Die hierzu notwendigen Produktarten geben wir mit folgender Definition an:

Definition 5.1 (Nachrichtenspezifikation)

Eine *Nachrichtenspezifikation* besteht aus

- einer abzählbaren Nachrichtenmenge X.

Wir unterscheiden die *partielle Spezifikation der Nachrichtenmenge N (N⊆)* und ihre *exakte Spezifikation (N)*:

- $[\![(X)_{\subseteq N}]\!] =_{def} \{sys \in SYS | X \subseteq N_{sys}\}$
- $[\![(X)_{N}]\!] =_{def} \{sys \in SYS | X = N_{sys}\}$

□

In Beispiel 5.6 legen wir fest, dass bestimmte Markierungen von Prozessereignissen eine Teilmenge der Nachrichtenmenge seien.

Als partielle beziehungsweise exakte Kanaltypisierung bezeichnen wir die beiden Produktarten zur Festlegung der auf einem gegebenen Kanal übertragbaren Nachrichten, die wir im Systemmodell durch die Abbildung *ctype* (siehe Definition 2.15) bestimmen:

Definition 5.2 (Kanaltypisierung)

Eine *Kanaltypisierung* besteht aus

- einem Kanalbezeichner c und
- einer abzählbaren Nachrichtenmenge X.

Die Semantik einer partiellen ($ct \subseteq$) und einer exakten (ct) Kanaltypisierung definieren wir durch

$$[\![(c,X)_{ct\subseteq}]\!] =_{def} \{sys \in SYS | c \in CH_{sys} \wedge X \subseteq N_{sys} \wedge X \subseteq ctype_{sys}(c)\},$$
$$[\![(c,X)_{ct}]\!] =_{def} \{sys \in SYS | c \in CH_{sys} \wedge X \subseteq N_{sys} \wedge ctype_{sys}(c) = X\}$$

□

5.1.2 Spezifikationsnetzwerke

Wie in Abschnitt 2.5.2 erwähnt, bilden wir im Systemmodell das zu entwickelnde Softwaresystem sowie dessen Umgebung durch das sogenannte Entwicklungs- beziehungsweise das Umgebungsnetzwerk ab. Entwicklungs- und Umgebungsnetzwerk bilden zusammen das sogenannte Realisierungsnetzwerk.

Neben dem Realisierungsnetzwerk enthält ein Entwicklungssystem noch eine endliche Menge von Spezifikationsnetzwerken, durch die wir die verschiedenen Sichten erfassen, die im Laufe einer Entwicklung zur Spezifikation des Realisierungsnetzwerkes erstellt werden. Die statische Struktur eines Spezifikationsnetzwerkes wird bestimmt durch dessen Komponentenmenge. Da zum einen ein Spezifikationsnetzwerk im allgemeinen schrittweise entwickelt wird, wobei schrittweise neue Komponenten hinzugenommen werden, und zum zweiten eine in bestimmte Ausschnitte strukturierte Dokumentation häufig sinnvoll ist, zum Beispiel ein Entwicklungsprodukt für jede Schicht einer Schichtenarchitektur, benötigen neben der ex-

akten Spezifikation der Komponentenmenge, eine Produktart um eine Teilmenge der Komponenten angeben zu können:

Definition 5.3 (Ausschnittsweise und exakte Spezifikation der Netzwerkkomponenten)

Eine *Spezifikation der Komponenten eines Spezifikationsnetzwerkes* besteht aus

- einem Netzwerkindex $i \in \mathbb{N}_+$ und
- einer endlichen Menge $X \subseteq KID$ von Komponentenbezeichnern.

Wir unterscheiden die *ausschnittsweise*($K \subseteq _snw$) und die *exakte Spezifikation* (K_snw):

- $[\![(i,X)_{K\subseteq_snw}]\!] =_{def} \{ sys \in SYS \mid i \in [1,...,(n_{snw})_{sys}] \wedge X \subseteq (K_i)_{sys} \}$
- $[\![(i,X)_{K_snw}]\!] =_{def} \{ sys \in SYS \mid i \in [1,...,(n_{snw})_{sys}] \wedge X = (K_i)_{sys} \}$

□

Neben der Festlegung der Komponenten eines Spezifikationsnetzwerkes ist auch die Angabe der Softwaresystemgrenze, das heißt die Unterscheidung zwischen zu entwickelnden Systemanteilen und Anteilen der (Softwaresystem-)Umgebung, eine Entwicklungsaufgabe. Im Systemmodell drücken wir diese Unterscheidung aus, indem wir die Komponenten eines Spezifikationsnetzwerkes als zu entwickelnd oder nicht zu entwickelnd klassifizieren. Die zu entwickelnden Komponenten eines Spezifikationsnetzwerkes zeichnen wir durch das Prädikat *develop* (vergleiche Definition 2.20) eines Systems aus. Durch die Produktart der Entwicklungsklassifikation können wir diese Eigenschaft einer Komponente formulieren:

Definition 5.4 (Entwicklungsklassifikation)

Eine *Entwicklungsklassifikation* besteht aus

- einem Komponentenbezeichner $k \in KID$ und
- einem booleschen Wert $b \in \{true, false\}$.

Die Semantik einer Entwicklungsklassifikation definieren wir durch

$$[\![(k,b)_d]\!] =_{def} \{ sys \in SYS \mid k \in K_{sys} \wedge develop_{sys}(k) = b \}$$

□

Da im allgemeinen nicht zu jedem Zeitpunkt einer Entwicklung klar ist, ob eine bestimmte Komponente Teil des zu entwickelnden Softwaresystems ist oder Teil der Umgebung, muss es möglich sein, die Komponenten eines Spezifikationsnetzwerkes anzugeben, ohne damit notwendigerweise auch die Entwicklungsklassifikation festzulegen. Mit den oben eingeführten Produktarten tragen wir dieser Anforderung Rechnung.

Zusätzlich zur Entwicklungsklassifikation einzelner Komponenten, definieren wir Produktarten, um die Komponenten des Entwicklungs- und den Umgebungsanteils eines Spezifikationsnetzwerkes *eindeutig/exakt* festlegen zu können:

Definition 5.5 (Spezifikation von Entwicklungs- und Umgebungsanteil eines Spezifikationsnetzwerkes)

Eine Spezifikation der Komponenten des Entwicklungs- beziehungsweise des Umgebungsanteils eines Spezifikationsnetzwerkes (bezeichnet mit *K_dnw* beziehungsweise *K_enw*) besteht aus

- einem Netzwerkindex $i \in \mathbb{N}_+$ und
- einer endlichen Menge $X \subseteq KID$ von Komponentenbezeichnern.

Den beiden Produktarten geben wir folgende Semantik[25]:

$$[\![(i,X)_{K_dnw}]\!] =_{def} \{ sys \in SYS \mid i \in [1,...,(n_{snw})_{sys}] \wedge X = (dK_i)_{sys} \}$$

$$[\![(i,X)_{K_enw}]\!] =_{def} \{ sys \in SYS \mid i \in [1,...,(n_{snw})_{sys}] \wedge X = (eK_i)_{sys} \}$$

□

Die Produktart der exakten Spezifikation des Entwicklungsanteils eines Spezifikationsnetzwerkes benötigen wir unter anderem als Baustein von Softwarearchitekturen. Die Produktart der *Softwarearchitektur* führen wir in Abschnitt 6.4 ein, wobei eine Softwarearchitektur den Zweck hat, alle Eigenschaften, das heißt sowohl statische Struktur als auch Verhaltenseigenschaften, einer Sicht auf das zu entwickelnde Softwaresystem eindeutig/exakt festzulegen.

Analog dazu dienen exakte Spezifikationen des Umgebungsanteils als Baustein von Umgebungsarchitekturen (siehe Abschnitt 6.4). Eine Umgebungsarchitektur ist beispielsweise ein Element einer Anforderungsspezifikation, wobei sie der Formulierung von Umgebungsannahmen dient.

5.1.3 Syntaktische Schnittstellen

Neben der Komponentenmenge ist die statische Struktur eines Netzwerkes durch die syntaktischen Schnittstellen der einzelnen Komponenten bestimmt, die durch folgende Produktarten spezifiziert werden:

Definition 5.6 (Schnittstellenspezifikation einer Komponente)

Eine *Schnittstellenspezifikation einer Komponente* besteht aus

- einem Komponentenbezeichner $k \in KID$ und

---

[25] siehe Definition 2.20 bezüglich dnw_i und enw_i

- zwei endlichen Mengen von Kanalbezeichnern $I, O \subseteq CH$

Wir unterscheiden die *ausschnittsweise*($SS\subseteq$) und die *exakte Spezifikation* (SS):

$$[\![(k,I,O)_{SS\subseteq}]\!] =_{def} \{sys \in SYS \mid k \in K_{sys} \wedge I \subseteq \Pi_1(comp_{sys}(k)) \wedge O \subseteq \Pi_2(comp_{sys}(k))\}$$

$$[\![(k,I,O)_{SS}]\!] =_{def} \{sys \in SYS \mid k \in K_{sys} \wedge I = \Pi_1(comp_{sys}(k)) \wedge O = \Pi_2(comp_{sys}(k))\}$$

□

Weiterhin ist die Spezifikation der syntaktischen Schnittstelle des zu entwickelnden Softwaresystems bedeutsam, etwa als Teil einer Anforderungsspezifikation (vergleiche Abschnitt 6.3). Die für diesen Zweck notwendigen Produktarten definieren wir analog zu Schnittstellenspezifikationen einzelner Komponenten (vergleiche Definition 5.6):

Definition 5.7 (Schnittstellenspezifikation des Entwicklungsnetzwerkes)

Eine *Schnittstellenspezifikation des Entwicklungsnetzwerkes* besteht aus

- zwei endlichen Mengen von Kanalbezeichnern $I, O \subseteq CH$

Wir unterscheiden die *ausschnittsweise*($SS\subseteq_dnw$) und die *exakte Spezifikation* (SS_dnw):

- $[\![(I,O)_{SS\subseteq_dnw}]\!] =_{def} \{sys \in SYS \mid I \subseteq in(dnw_{sys}) \wedge O \subseteq out(dnw_{sys})\}$
- $[\![(I,O)_{SS_dnw}]\!] =_{def} \{sys \in SYS \mid I = in(dnw_{sys}) \wedge O = out(dnw_{sys})\}$

□

Die exakte Schnittstellenspezifikation des Entwicklungsnetzwerkes ist beispielsweise Bestandteil einer Anforderungsspezifikation (siehe Abschnitt 6.3), da dadurch die statische Struktur der Außensicht (Black-Box-Sicht) des Softwaresystems eindeutig beschrieben ist. Die Außensicht entspricht dem Betrachtungsstandpunkt eines Anwenders des Softwaresystems und ist somit passend zum Zweck einer Anforderungsspezifikation.

In Beispiel 5.1 illustrieren wir die Verwendung der oben eingeführten Produktarten, indem wir ein Strukturdiagramm als Produktmenge interpretieren.

Beispiel 5.1 (Ausschnitt der statischen Struktur des Bibliotheksanwendungssystems)

In der informellen Beschreibung des Entwicklungsziels für unser Bibliothekssystem (siehe Abschnitt 1.7) werden Leser und Bibliotheksangestellte als Anwendergruppen/-rollen angegeben, die das zu entwickelnde Softwaresystem auf spezifische Weise, zum Beispiel mit spezifischen Rechten, nutzen. In einer ersten Spezifikation modellieren wir die beiden Rollen durch Komponenten, die wir *Leser* und *BibAngest* nennen (siehe Strukturdiagramm in Abbildung 5.3). Für die unterschiedlichen Anwenderrollen bietet das Softwaresystem geeignete Schnittstellen. In der statischen Struktur zeigt sich dies durch Kanäle zwischen den die Rollen abbildenden Komponenten und dem Entwicklungsnetzwerk.

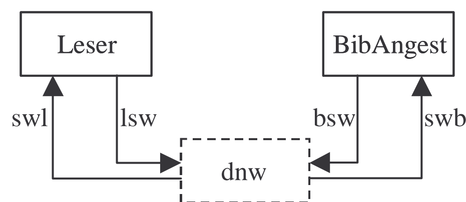


Abbildung 5.3:Rollen *Leser* und *BibAngest* und ihre Kommunikationsverbindungen zu dem zu entwickelnden Netzwerk *dnw*.

Ohne Beschränkung der Allgemeinheit erfassen wir die Spezifikation durch das mit 1 indizierte Spezifikationsnetzwerk, das heißt durch snw_1. Damit verstehen wir das Strukturdiagramm aus Abbildung 5.3 als graphische Darstellung folgender Produktmenge:

$$\begin{aligned}\{&(1, \{\text{BibAngest, Leser}\})_{K\subseteq_snw},\\ &(\text{BibAngest}, \text{false})_d,\\ &(\text{Leser}, \text{false})_d,\\ &(\{\text{lsw}, \text{bsw}\}, \{\text{swl}, \text{swb}\})_{SS\subseteq_dnw},\\ &(\text{Leser}, \{\text{swl}\}, \{\text{lsw}\})_{SS\subseteq},\\ &(\text{BibAngest}, \{\text{swb}\}, \{\text{bsw}\})_{SS\subseteq}\}\end{aligned}$$

□

5.1.4 Topologische Klassifikation

Die in Form von Spezifikationsnetzwerken gegebenen Sichten auf das Realisierungsnetzwerk, und damit das Softwaresystem und seine Umgebung, dienen im allgemeinen unterschiedlichen Zwecken.

So kann ein Spezifikationsnetzwerk zur Spezifikation der Außensicht des Softwaresystems eingesetzt werden, ohne damit Anforderungen an die interne Struk-

tur/Topologie zu verbinden. Die Komponenten des Spezifikationsnetzwerkes dienen in diesem Fall lediglich als modulare Einheiten, um die Spezifikation der Außensicht zu strukturieren. Andererseits kann aber ein Spezifikationsnetzwerk auch dazu dienen, die Topologie des Softwaresystems bis auf Isomorphie eindeutig festzulegen. Das bedeutet, dass wir, je nach Zweck eines Spezifikationsnetzwerkes, mit dessen Komponenten unterschiedliche topologische Anforderungen verbinden, was wir durch die, in Abschnitt 2.5.3 eingeführte, topologische Klassifikation von Komponenten ausdrücken können. Definition 5.8 gibt die zugehörige Produktart an:

Definition 5.8 (Topologische Klassifikation)

Eine *topologische Klassifikation* besteht aus

- einem Komponentenbezeichner $k \in KID$ und
- einer topologischen Klasse $x \in TClass$[26].

Die Semantik einer Topologietypisierung definieren wir durch

$$[\![(k, x)_{tt}]\!] =_{def} \{ sys \in SYS | k \in K_{sys} \wedge tclass(comp_{sys}(k)) = x \}^{27}$$

□

Das folgende Beispiel illustriert eine topologische Klassifikation in Verbindung mit der Spezifikation statischer Struktur, wobei wir als Notation ein Strukturdiagramm verwenden. Zudem werden Komponenten als Entwicklungsanteil klassifiziert.

Beispiel 5.2 (Statische Struktur mit topologischer und Entwicklungs-Klassifikation)

Wir setzen an dieser Stelle unser Bibliotheksbeispiel fort, indem wir eine (erste) Struktur für die Beschreibung des Bibliotheks-Softwaresystems angeben. Komponenten verstehen wir hierbei als rein konzeptuelle Einheiten, mit denen wir keine topologischen Vorgaben für die Implementierung des Softwaresystems machen. Diese Rolle von Komponenten ist beispielsweise typisch für frühe Entwurfsphasen in einem Forward-Engineering-Vorgehen, wo es (lediglich) darum geht, die Gesamtaufgabe eines Systems durch Aufteilung in Teilaufgaben zu erfassen (divide et impera). Dementsprechend klassifizieren wir alle Komponenten als *behavior*-Komponenten (vergleiche gegebenenfalls Abschnitt 2.5.3).

Das Strukturdiagramm in Abbildung 5.4 zeigt die Struktur des Bibliothekssystems, wobei die topologische Klassifikation als behavior-Komponente durch die Zeichenfolge *::B* einer Komponentenmarkierung dargestellt ist.

---

[26] siehe Definition 2.17 bezüglich *TClass*

[27] *tclass(y)* bezeichnet die Projektion auf die topologische Klasse einer Topologiekomponente *y* (siehe Definition 2.17); die Abbildung *comp* ordnet jedem Komponentenbezeichner eine Topologiekomponente zu (siehe Definition 2.19)

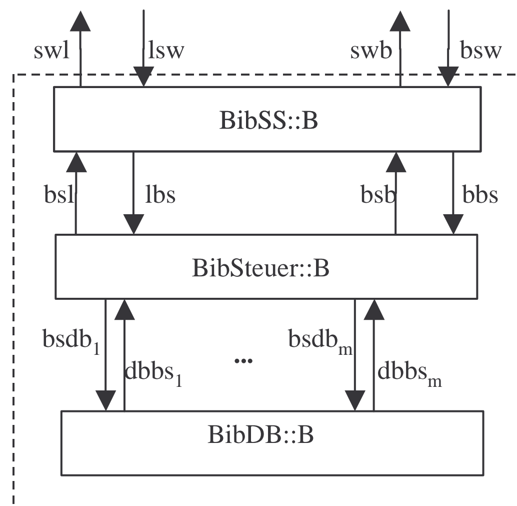


Abbildung 5.4: Schichtenarchitektur des Entwicklungsnetzwerkes

Wir gehen von einer klassischen 3-Schichten-Architektur für Informationssysteme aus (siehe zum Beispiel [DW99a]), in der

- eine Datenhaltungsschicht der persistenten und konsistenten Datenhaltung dient (repräsentiert durch die *BibDB* genannte Komponente),
- die darauf aufbauende Steuerungsschicht (Komponente *BibSteuer*) die Steuerung von Abläufen übernimmt, und
- die Schnittstellenschicht (Komponente *BibSS*) die Aufgabe der Interaktion mit den Anwender regelt.

Ohne Beschränkung der Allgemeinheit wählen wir, wie in Beispiel 5.1, das mit 1 indizierte Spezifikationsnetzwerk zur Abbildung der Schichtenarchitektur(-sicht), so dass wir das Strukturdiagramm aus Abbildung 5.4 durch folgende Menge von Entwicklungsprodukten, die wir mit *BibStat1* bezeichnen, interpretieren:

$$
\begin{aligned}
\text{BibStat1} =_{\text{def}} \{ & (1, \{\text{BibSS, BibSteuer, BibDB}\})_{K\subseteq}, \\
& (\text{BibSS, true})_d, (\text{BibSteuer, true})_d, (\text{BibDB, true})_d, \\
& (\text{BibSS,behavior})_{tt}, (\text{BibSteuer,behavior})_{tt}, (\text{BibDB,behavior})_{tt}, \\
& (\text{BibSS}, \{\text{lsw}\}, \{\text{swl}\})_{SS\subseteq}, \\
& (\text{BibSteuer}, \{\text{lbs, bbs}\}, \{\text{bsl, bsb}\})_{SS\subseteq}, \\
& (\text{BibDB}, \{\text{bsdb}_1, \dots, \text{bsdb}_m\}, \{\text{dbbs}_1, \dots, \text{dbbs}_m\})_{SS\subseteq} \}
\end{aligned}
$$

□

5.2 Netzwerk-Verhalten

In unserem Systemmodell erfassen wir das zu entwickelnde Softwaresystem und dessen Umgebung in Form eines Komponentennetzwerkes. Mit den sogenannten Spezifikationsnetzwerken enthält das Systemmodell zusätzlich Elemente, durch die wir, ebenfalls in Form von Netzwerken, Eigenschaften von Softwaresystem und Umgebung angeben können.

In diesem Abschnitt betrachten wir Entwicklungsprodukte zur Charakterisierung komponentenübergreifender Verhaltenseigenschaften von Netzwerken. Zwar legen die Eigenschaften der einzelnen Komponenten eines Netzwerkes dessen Eigenschaften auf eindeutige Weise fest (Kompositionalität), so dass es theoretisch ausreichen würde, nur Produktarten zur Spezifikation *einzelner* Komponenten anzugeben. Aus methodischer Sicht halten wir jedoch einen komponentenübergreifenden Blickwinkel für sinnvoll.

Grundsätzlich kann aus diesem Blickwinkel dargestellt werden, wie Komponenten zusammenwirken, um umfassende Aufgaben zu realisieren. Dies ist insbesondere notwendig, um die Realisierung fachlicher Aufgaben durch Softwaresystem und Umgebung darzustellen, beispielsweise die Realisierung von Geschäftsprozessen. Grundsätzlich hilft die Spezifikation zentraler Kooperationsformen zwischen Komponenten dabei, die Rolle, die einzelne Komponenten einnehmen, verständlich zu machen.

Im Forward Engineering ist die Spezifikation (einiger zentraler) komponentenübergreifender Verhaltenseigenschaften häufig vorbereitend für die (vollständige) Verhaltensspezifikation der einzelnen Komponenten.

Der komponentenübergreifende und damit netzwerk-globale Blickwinkel auf Netzwerkverhalten ist ähnlich zu dem komponentenunabhängigen und damit globalen Verhaltensmodell der Prozesse der ganzheitlichen Perspektive unseres Systemmodells. Dies wird auch aus den Perspektiven-Zusammenhängen deutlich:

Durch eine Interaktionszuordung hi_i ordnen wir Prozessereignissen Ausführungsfolgen des Spezifikationsnetzwerkes snw_i zu, die wir als korrekte Realisierungen des Prozessereignisses verstehen. Mittels hi_i können wir jede Eigenschaft von Prozessverhalten in eine Eigenschaft von Netzwerkverhalten übersetzen. Da wir fordern, dass jedes Spezifikationsnetzwerk eine Realisierung der fachlichen Aufga-

benstellung zu beschreiben hat, ergeben sich mittels hi_i aus den Eigenschaften des Prozessverhaltens die (minimalen) Anforderungen an das Netzwerkverhalten.

Die Forderung, dass ein Spezifikationsnetzwerk eine Realisierung der fachlichen Aufgabenstellung darzustellen hat, haben wir bezüglich des Verhaltensaspekts im Systemmodell durch die Forderung

$exec(snw_i) \subseteq GStreams_i$ (vergleiche Definition 2.30)

erfasst. Dies bedeutet beispielsweise im Falle einer Reaktionseigenschaft

„a führt immer zu (einer der Alternativen aus) X“

(wobei a eine Ereignismarkierung sei und X eine Menge von Prozess-Isomorphieklassen) die etwa durch eine Geschäftsprozessspezifikation $(a,X)_{reak}$ (vergleiche Definition 4.1) ausgedrückt werden kann, dass für ein Spezifikationsnetzwerk snw_i

$hi_i(a)$ führt zu $hi_i(X)$[28]

zu gelten hat.

Die im Systemmodell erfassten Zusammenhänge, zusammen mit den in Abschnitt 4.1 festgelegten Produktarten für die Spezifikation von (fachlichem) Prozessverhalten, legen die im Folgenden gegebenen Produktarten zur Verhaltensspezifikation von Netzwerken nahe. Jede Form von Netzwerkspezifikation hat hierbei ein Pendant zur Prozessspezifikation. Abbildung 5.5 gibt einen Überblick der Produktarten zur Spezifikation von Netzwerk- und Komponenten-Verhalten, wobei wir letztere in den Abschnitten 5.3 und 5.4 behandeln.

[28] um genau zu sein, müssen wir anstelle von $hi_i(X)$ schreiben:

$$\left\{\varphi \in (IS_i)^\omega \,\middle|\, \exists p \in X, \psi \in (M \rightarrow \mathbb{N})^\omega . \; \psi \in cexecutions(Step(p)) \wedge \varphi \in is_flatten^*(\psi, hi_i)\right\}$$

und damit analog zu Definition 2.29 aus X die Menge der IS_i-Ströme via Step-Semantik und hi_i ableiten.

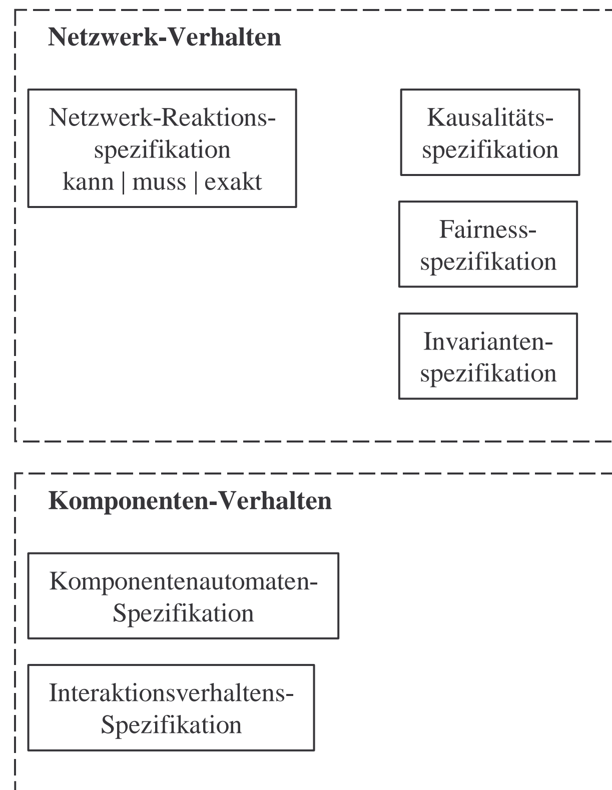


Abbildung 5.5: Produktarten zur Spezifikation von Netzwerk- und Komponenten-Verhalten.

Eine verbreitete Notation für die Spezifikation globaler Verhaltenseigenschaften sind Sequenzdiagramme (Message Sequence Charts), deren methodischer Einsatz umfassend in [Krü01] behandelt wird und die wir auch in den folgenden Beispielen einsetzen. In [Krü01] wird das Systemverhalten ähnlich zu unserer komponentenbasierten Perspektive modelliert. Auf notationsunabhängige Weise wird in [Web91] eine Methodik vorgestellt, in der die Spezifikation globaler Sicherheits- und Lebendigkeitseigenschaften ein zentraler Bestandteil ist.

5.2.1 Reaktionsspezifikation

Aus fachlicher Sicht ist die Spezifikation von Geschäftsprozessen eine wichtige Entwicklungsaufgabe. Wie in Abschnitt 4.1 gezeigt, können wir Geschäftsprozesse als Reaktionseigenschaften eines (Anwendungs-)Systems verstehen. Daher benötigen wir, analog zu den Produktarten für Prozess-Reaktionseigenschaften aus Abschnitt 4.1, Produktarten zur Spezifikation von Netzwerk-Reaktionseigenschaften, um beschreiben zu können, wie Geschäftsprozesse durch das Softwaresystem-Umgebungs-Netzwerk realisiert werden:

Definition 5.9 (Reaktionsspezifikation von Spezifikationsnetzwerken)

Sei CH´ eine endliche Menge von Kanälen und S´ eine abzählbare Menge von Zuständen. Eine Reaktionsspezifikation der Ausführungsfolgen eines Spezifikationsnetzwerkes snw_i, $i \in [1,...,n_{snw}]$, ist ein 3-Tupel (i, t, R) bestehend aus

- einem Index $i \in \mathbb{N}_+$,
- einer Menge endlicher (Trigger-)Abläufe $T \in \wp\left(IS_{CH',S'}{}^{*}\right)$ und
- einer endlichen Menge von Reaktionsabläufen $R \in \wp\left(IS_{CH',S'}{}^{\omega}\right)$.

Wir unterscheiden drei Arten von Reaktionsspezifikationen

- *kann-Reaktionsspezifikation/Reaktionsszenarien (kann_isreak)*
- *muss-Reaktionsspezifikation (muss_isreak)* und
- *exakte Reaktionsspezifikation (isreak).*

Die Semantik der drei Arten von Reaktionsspezifikationen definieren wir wie folgt:

- $$\llbracket (i,T,R)_{kann_isreak} \rrbracket =_{def} \left\{ sys \in SYS \middle| i \in \left[1,...,(n_{snw})_{sys}\right] \wedge \forall t \in T.R \subseteq (isreak_i)_{sys}(t) \right\}$$
- $$\llbracket (i,T,R)_{muss_isreak} \rrbracket =_{def} \left\{ sys \in SYS \middle| i \in \left[1,...,(n_{snw})_{sys}\right] \wedge \forall t \in T.(isreak_i)_{sys}(t) \subseteq R \right\}$$
- $$\llbracket (i,T,R)_{isreak} \rrbracket =_{def} \left\{ sys \in SYS \middle| i \in \left[1,...,(n_{snw})_{sys}\right] \wedge \forall t \in T.(isreak_i)_{sys}(t) = R \right\}$$

□

Eine typische Anwendung von Netzwerk-Reaktionsszenarien ist die Skizzierung der Rolle/Aufgaben von Netzwerk-Komponenten sowie deren Zusammenwirken anhand einiger exemplarischer Interaktionsfolgen, durch die bestimmte Geschäftsprozesse realisiert werden. In einem Forward-Engineering Vorgehen ist die Entwicklung von Reaktionsszenarien beispielsweise ein Entwicklungsschritt, der sich an die Spezifikation der Geschäftsprozesse und einer Netzwerk-Struktur anschließt, und als Vorbereitung für die Spezifikation des Verhaltens der einzelnen Netzwerk-Komponenten dient. In folgendem Beispiel betrachten wir die Realisierung einer Vormerkungsbearbeitung in unserem Bibliothekssystem.

Reaktionsspezifikationen von Netzwerken dienen auch dazu, die *minimalen* Anforderungen, die sich aus Prozess-Reaktionsspezifikationen via Step-Semantik und Interaktionszuordnung hi_i (wie eingangs von Abschnitt 5.2 beschrieben) ergeben, gegebenenfalls zu verstärken. Ein Beispiel hierfür finden wir in Beispiel 5.3, wo wir festlegen, dass alle drei Datenbank-Interaktionen (*m.vormerken*, m.eqstatus* und *m.zuteilen*), die im Rahmen einer Vormerkungsbearbeitung auszuführen sind, auf denselben Kanalpaaren, $bsdb_k$ und $dbbs_k$, stattzufinden haben.

Beispiel 5.3 (Reaktionsszenario/kann-Reaktionsspezifikation eines Spezifikationsnetzwerkes)

In diesem Beispiel geben wir ein Reaktionsszenario des Bibliothekssystems an, wobei wir uns auf die in Beispiel 5.2 gegebene Schichtenarchitektur beziehen, in der das Bibliothekssystem in die drei Komponenten *BibSS*, *BibSteuer* und *BibDB* gegliedert ist. Der Zweck des Reaktionsszenarios ist es, die Rolle der Komponente *BibSteuer* anhand einer exemplarischen Interaktionsfolge zu veranschaulichen. Dabei betrachten wir eine Interaktionsfolge, durch welche die Bearbeitung eines Vormerkungswunsches realisiert wird.

Die Aufgabe der Komponente *BibSteuer* ist die Koordinierung der einzelnen Bearbeitungsschritte fachlicher Abläufe. Die Anforderungen an diese Koordinierungsaufgabe ergeben sich aus der Kausalrelation der zugehörigen Geschäftsprozessspezifikationen (siehe Beispiel 4.1 für den Prozess der Vormerkungsbearbeitung). Das Reaktionsszenario wählen wir so, dass es den Fall einer Vormerkungsbearbeitung abbildet, in dem das gewünschte Medium verfügbar ist und somit dem Leser (sofort) zugeteilt werden kann. Abbildung 5.6 enthält ein Sequenzdiagramm zur graphischen Darstellung der Interaktionsfolge des Szenarios, welche die Reaktion auf die Äußerung eines Vormerkungswunsches darstellt.

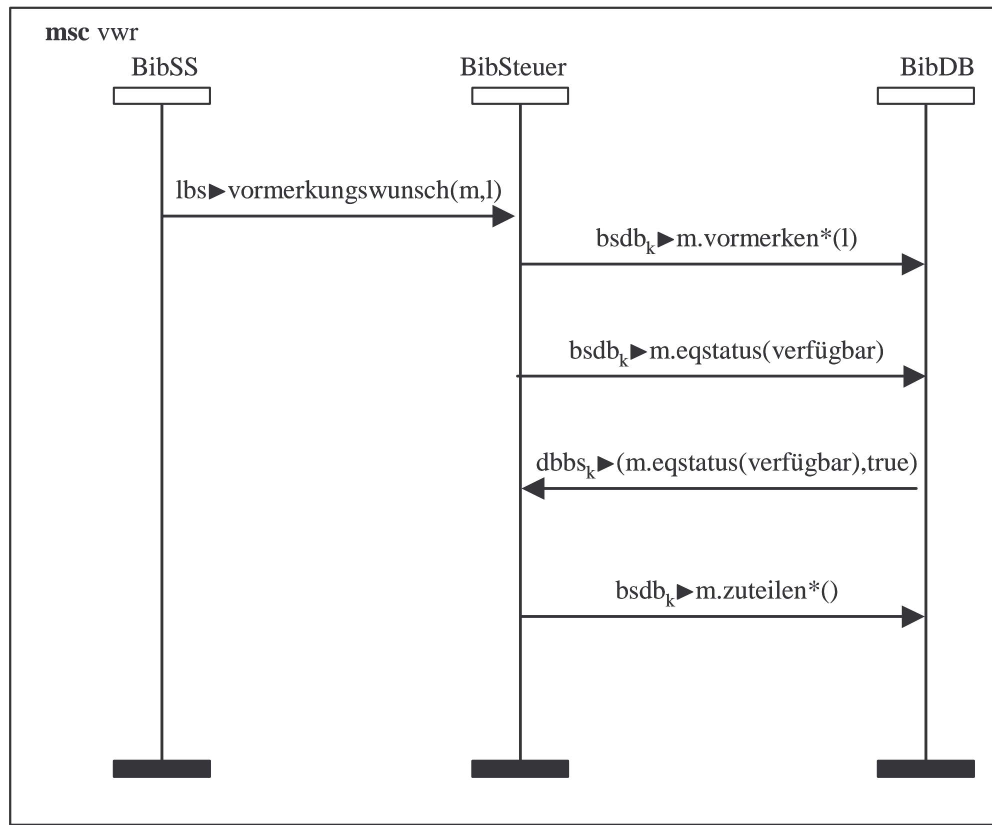


Abbildung 5.6: Eine der möglichen Interaktionsfolgen zur Realisierung einer Vormerkungsbearbeitung.

Für die Realisierung der Zustandsbeobachtungen und der Zustandsänderungen, die im Rahmen einer Vormerkungsbearbeitung durchzuführen sind, ist es von Bedeutung, wie der fachliche Zustandsraum durch die Komponenten des betrachteten (Spezifikations-)Netzwerkes realisiert wird. Wir gehen in diesem Beispiel davon aus, dass der gesamte fachliche Zustandsraum durch die Datenbank-Komponente *BibDB* realisiert wird (vergleiche Beispiel 5.5, wo wir diese Anforderung mathematisch angeben). Zustandsbeobachtungen und –änderungen realisieren wir daher durch entsprechende Interaktion mit der Datenbank-Komponente. Hierzu definieren wir die Menge der Zustandsbeobachtungen Z und der Zustandsübergänge T als Teilmengen der Nachrichtenmenge N, so dass etwa durch das senden einer Nachricht $t \in T$ eine Änderung des Zustands von *BibDB* gemäss *t* impliziert wird (siehe hierzu Beispiel 5.7, wo wir die Realisierung von zustandsbezogenenen Ereignismarkierungen spezifizieren, und Beispiel 5.5, wo wir durch einen Automaten das Verhalten von *BibDB* festlegen). Auf Zustandsabfragen $z \in Z$ reagiere *BibDB* mit der Rückgabe einer Nachricht, die ein Zwei-Tupel *(z,b)* ist, bestehend aus der Abfrageoperation *z* sowie dem Wert, den *z* angewandt auf den „aktuellen" Zustand von *BibDB* ergibt (vergleiche Beispiel 5.5). Vereinfachend fordern wir, dass die Menge der Ereignismarkierungen M eine Teilmenge der Nachrichten N seien.

Ausgelöst wird eine Vormerkungsbearbeitung durch die Äußerung eines Vormerkungswunsches. In Beispiel 5.6 legen wir die entsprechenden Interaktionsfolgen fest.

Das unten gegebene Entwicklungsprodukt *VMSW1* umfasst neben den Reaktionsszenarien Entwicklungsprodukte, um die oben beschriebenen Anforderungen, welche den Kontext der Szenarien bestimmen, zu erfassen. Wir beziehen uns dabei, wie erwähnt, auf Produkte aus einer Reihe anderer Beispiele:

- Sei MEDIUM´ die algebraische Spezifikation aus Beispiel 3.2, durch welche wir die fachliche Entitätssignatur $\Sigma_{MEDIUM'}$ und Rechenstruktur-Klasse $C_{MEDIUM'}$ charakterisieren und damit unter anderem die Sorte *Medium* sowie die Operation

  $\text{vormerken} : \text{Medium} \times \text{id}_{\text{Leser}} \rightarrow \text{Medium}$

  und

  $\text{zuteilen} : \text{Medium} \rightarrow \text{Medium}$

  einführen.

- Sei weiterhin ACT die algebraische Spezifikation aus Beispiel 4.1, welche MEDIUM´ erweitert um

  $\text{vormerkungswunsch} : \text{id}_{\text{Medium}} \times \text{id}_{\text{Leser}} \rightarrow \text{Act}$

  sowie die Operation

  $\text{eqstatus} : \text{Medium} \times \text{Status} \rightarrow \text{Bool}$

- Sei zudem *BibStat1* das Entwicklungsprodukt aus Beispiel 5.2, mit dem die statische Struktur des Spezifikationsnetzwerkes snw_1 festgelegt wird.

Damit sei das Produkt *VMSW1* wie folgt definiert:

$$\text{VMSW1} =_{\text{def}} (\text{p})_{\text{pred}}$$

mit

$$\text{p} : \text{SYS} \rightarrow \mathbb{B}$$

wobei für alle $\text{sys} \in \text{SYS}$ gelte:

$$\begin{aligned}
\text{p}(\text{sys}) \Leftrightarrow_{\text{def}} \; & \text{sys} \in \left[\!\left[(\{\text{Medium}, \text{Leser}, \text{Status}\}\Sigma_{\text{ACT}}, C_{\text{ACT}})_{\text{muss_red}} \right]\!\right] \wedge \\
& \text{sys} \in [\![\text{BStat1}]\!] \wedge \\
& \text{sys} \in \left[\!\left[(\text{M}_{\text{sys}})_{\subseteq \text{N}} \right]\!\right] \wedge \\
& \text{sys} \in \left[\!\left[(\text{Reply}, \text{N})_{\subseteq \text{N}} \right]\!\right] \wedge \\
& \forall \text{m} \in (\text{A}_{\text{sys}})_{\text{id}_{\text{Medium}}}, (\text{A}_{\text{sys}})_{\text{id}_{\text{Leser}}} . \\
& \quad \text{sys} \in \left[\!\left[\left(1, (\text{hi}_1)_{\text{sys}}(\text{vormerkungswunsch}(\text{m}, \text{l})), \text{vwr}(\text{m}, \text{l})\right)_{\text{is_reak}\supseteq} \right]\!\right] \}
\end{aligned}$$

wobei

Reply die Menge der „Rückgabenachrichten“ von BibDB wie in Beispiel 5.5 gegeben sei.

Eigenschaften von $(hi_1)_{sys}(\text{vormerkungswunsch}(m,l))$, das heißt den Interaktionsfolgen zur Realisierung eines Vormerkungswunsches für ein Medium *m* und einen Leser *l*, legen wir in Beispiel 5.6 fest.

Mit *vwr(m,l)*,für ein Medium *m* und einen Leser *l*, bezeichnen wir Menge von Ausführungsfolgen, welche durch das Sequenzdiagramm in Abbildung 5.6 graphisch dargestellt ist:

$$\text{vwr}: (A_{sys})_{id_{Medium}} \times (A_{sys})_{id_{Leser}} \rightarrow \wp\left(((IS_1)_{sys})^{\omega}\right)$$

Für alle $m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}}$ gelte:

$$\begin{aligned}
\text{vwr}(m,l) =_{def} \Big\{ \varphi \in \left(((IS_1)_{sys})^{\omega}\right) \Big| & \exists j_1, j_2, ..., j_5 \in \mathbb{N}_+, k \in [1,...,m]. \\
& (j_1 < j_2 <...< j_5) \wedge \\
& \text{in}\left(\text{vormerkungswunsch}^{A_{sys}}(m,l), (\Pi_2(\varphi @ j_1)(\text{lbs}))\right) \wedge \\
& \text{in}\left(\text{m.vormerken}*(1)^{FS_{sys}}, (\Pi_2(\varphi @ j_2)(\text{bsdb}_k))\right) \wedge \\
& \text{in}\left(\text{m.eqstatus}(\text{verfügbar})^{FS_{sys}}, (\Pi_2(\varphi @ j_3)(\text{bsdb}_k))\right) \wedge \\
& \text{in}\left(\left(\text{m.eqstatus}(\text{verfügbar})^{FS_{sys}}, \text{true}^{FS_{sys}}\right), (\Pi_2(\varphi @ j_4)(\text{dbbs}_k))\right) \wedge \\
& \text{in}\left(\text{m.zuteilen}^*(\ )^{FS_{sys}}, (\Pi_2(\varphi @ j_5)(\text{bsdb}_k))\right) \Big\}
\end{aligned}$$

□

5.2.2 Kausalität und Fairness

Reaktionsspezifikationen machen keine Aussage über das zahlenmäßige Verhältnis von Auslöser und Reaktionen. Eine typische Verstärkung von Reaktionsspezifikationen besteht darin, zu fordern, dass die Anzahl der Auslöser mit der Anzahl der Reaktionen (in einem System-Lebenszyklus) übereinstimmt, was unter dem Begriff der strikten Kausalität (zwischen Auslöser und Reaktion) bekannt ist. Als Grundlage für eine zur Kausalitätsspezifikation von Prozessen (vergleiche Definition 4.2) verwandte Produktart zur Kausalitätsspezifikation von Spezifikationsnetzwerken, führen wir zunächst die Hilfsfunktion #* ein:

Funktion #*

Sei Y eine Menge. Für Ströme über Y definieren wir

$$\#^*: \wp(Y^{\omega}) \times Y^{\omega} \rightarrow \mathbb{N}$$

wobei für alle $X \in \wp(Y^{\omega}), \varphi \in Y^{\omega}$ gelte:

$$\#^*(X, \varphi) =_{def} |\{i \in \mathbb{N} | \exists x \in X. x \sqsubseteq \varphi \uparrow i\}|$$

#* liefert die Anzahl der Vorkommen von Strömen aus X in einem Strom φ. Zu beachten ist hierbei, dass, falls X sowohl einen Strom x wie auch einen in x enthaltenen Teilstrom umfasst, ein Vorkommen von x in φ mehrfach in #* eingeht, da auch der von x umfasste Teilstrom gezählt wird.

□

Definition 5.10 (Kausalitätsspezifikation von Ausführungsfolgen)

Sei CH´ eine endliche Menge von Kanälen und S´ eine abzählbare Menge von Zuständen. Eine *Kausalitätsspezifikation* ist ein 3-Tupel (i, T , X) bestehend aus

- einem Index $i \in \mathbb{N}_+$,
- einer Menge von(Interaktions-Zustands-)Strömen $T \in \wp\left((IS_{CH',S'})^{\omega} \right)$ und
- einer Menge von (Interaktions-Zustands-)Strömen $X \in \wp\left((IS_{CH',S'})^{\omega} \right)$.

Die Semantik einer Kausalitätsspezifikation definieren wir wie folgt:

$$[\![(i,T,X)_{iskausal}]\!] =_{def} \{ sys \in SYS | i \in \left[1,\ldots,(n_{snw})_{sys}\right] \wedge \forall \psi \in exec\left((snw_i)_{sys} \right) . \left(\#^*(T,\psi) = \#^*(X,\psi) \right) \}$$

□

Um in Kombination mit einer Reaktionsspezifikation in der Menge der zulässigen Reaktionsalternativen die Menge der „erwünschten“ Alternativen auszeichnen zu können, führen wir analog zu Definition 4.3 die Fairnessspezifikation von Netzwerk-Ausführungsfolgen ein:

Definition 5.11 (Fairnessspezifikation von Ausführungsfolgen)

Sei CH´ eine endliche Menge von Kanälen und S´ eine abzählbare Menge von Zuständen. Eine *Fairnessspezifikation* ist ein 2-Tupel (T , X) bestehend aus

- einem Index $i \in \mathbb{N}_+$,
- einer Menge von(Interaktions-Zustands-)Strömen $T \in \wp\left((IS_{CH',S'})^{\omega} \right)$ und
- einer Menge von (Interaktions-Zustands-)Strömen $X \in \wp\left((IS_{CH',S'})^{\omega} \right)$.

Die Semantik einer Fairnessspezifikation definieren wir wie folgt:

$$[\![(i,T,X)_{issfair}]\!] =_{def} \{ sys \in SYS | i \in \left[1,\ldots,(n_{snw})_{sys}\right] \wedge \forall \psi \in exec\left((snw_i)_{sys} \right) . \left(\#^*(T,\psi) = \infty \right) \Rightarrow \left(\#^*(X,\psi) = \infty \right) \}$$

□

Als Randbedingung für die sinnvolle Verknüpfung von Reaktionsspezifikationen mit Kausalitäts- und Fairnessspezifikationen ergibt sich für

$(i,T,Y)_{*_isreak} \wedge (i,T,X)_{iskausal}$

und

$(i,T,Y)_{*_isreak} \wedge (i,T,X)_{issfair}$

dass gelte $X \subseteq Y$.

5.2.3 Invariantenspezifikation

Ergänzend zu obigen Produktarten zur Spezifikation von Lebendigkeitseigenschaften, führen die Invariantenspezifikation für Netzwerk-Ausführungsfolgen ein, um Sicherheitseigenschaften angeben zu können (vergleiche Abschnitt 4.2, wo wir Invariantenspezifikationen für Prozesse behandeln).

Definition 5.12 (Invariantenspezifikation)

Sei CH´ eine endliche Menge von Kanälen und S´ eine abzählbare Menge von Zuständen. Eine Invariantenspezifikation besteht aus

- einem Index $i \in \mathbb{N}_+$ und
- einem Prädikat $Q : (IS_{CH´,S´})^{\omega} \to \mathbb{B}$.

Die Semantik einer Invariantenspezifikation definieren wir wie folgt:

$$[\![(i,Q)_{isinv}]\!] =_{def} \{ sys \in SYS \mid i \in [1,...,(n_{snw})_{sys}] \wedge \\ \forall \psi \in exec((snw_i)_{sys}), \psi´ \in ((IS_i)_{sys})^{\omega} . \\ \psi´ \sqsubseteq \psi \Rightarrow Q(\psi´) \}$$

□

5.3 Automatensicht einer Komponente

Die in Abschnitt 5.2 eingeführten Produktarten stellen verschiedene Formen axiomatischer Verhaltensspezifikationen dar. Sie eignen sich für abstrakte Verhaltensbeschreibungen, die frei von unnötigen Details sind und so Überspezifikation vermeiden helfen. Dadurch kann die unerwünschte Beeinflussung von Realisierungsentscheidung verhindert werden.

Aufgrund der Abstraktheit axiomatischer Spezifikationen ist es jedoch häufig schwierig, die Korrektheit einer operationellen Realisierungsbeschreibung, zum Beispiel mittels einer prozeduralen Programmiersprache, bezüglich einer axiomatischen Spezifikation zu zeigen. Dies hängt damit zusammen, dass in der axiomati-

schen Spezifikation Eigenschaften von „Gesamtverhalten“ festgelegt werden, während in einer operationellen Beschreibung Verhalten schrittweise erzeugt wird, durch Folgen von Operationen.

Ein weiterer Schwachpunkt axiomatischer Spezifikationen ist das Fehlen eines einfach nachprüfbaren Vollständigkeitskriteriums für eine Spezifikation.

Um die konzeptuelle Lücke zwischen axiomatischer Spezifikation und operationeller (Beschreibung einer) Realisierung zu schließen, sind Verhaltensbeschreibungen durch Automaten ein geeignetes Mittel (siehe Abschnitt 2.5.5). In unserem Systemmodell setzen wir Automaten zur Festlegung der Verhaltensfunktionen und der Ausführungsfolgen von Spezifikationskomponenten ein. Die entsprechenden Systemmodellelemente sind in der sogenannten Automatensicht einer Komponente zusammengefasst (siehe Definition 2.22 und Definition 2.23).

Im Folgenden gehen wir auf die Entwicklungsproduktart ein, die der Festlegung der Automatensicht von Spezifikationskomponenten dient. Eine Automatenspezifikation umfasst alle Elemente eines Automaten gemäss Definition 8.27, unter anderem eine Zustandsmenge und eine Transitionsfunktion. Bezüglich dieser beiden Elemente und der Interpretation eines Produktes sind zwei Aspekte von zentraler Bedeutung:

- der Zusammenhang zwischen den Zuständen des Entwicklungsproduktes und den Zuständen des Systems, das heißt der entsprechende Komponente
- der Zusammenhang zwischen den Transitionsfunktionen von Entwicklungsprodukt und Komponente, insbesondere der Interpretation partieller Automaten des Produktes

Wir gehen nun auf diese beiden Aspekte ein.

Spezifikations- und Komponentenzustände

Im allgemeinen kann die Zustandsmenge einer Komponente beziehungsweise eines Systems sehr groß sein. Um mit endlichen Mitteln und auf überschaubare Weise große oder sogar unendliche Zustandsmengen beschreiben zu können, ist es innerhalb der Spezifikation daher häufig notwendig, Klassen von Zuständen zu bilden. In Notationen zur Spezifikation von Automaten, die das Verhalten von Komponenten charakterisieren, werden zu diesem Zweck Automatenzustände mit Prädikaten über Komponenten annotiert (siehe zum Beispiel State-Transition-Diagrams der UML [OMG01] oder der in [Rum96] eingeführten Syntax für Automatenspezifikationen, die weitreichende Wohlgeformtheitskriterien für Automatenzustände umfasst). Es wird also zwischen den Zuständen in der Automatenspezifikation und den Zuständen, die das betrachtete System beziehungsweise die Komponente einnehmen kann, unterschieden, wobei Zustände der Automatenspezifikation für Klassen von Komponentenzuständen stehen. Beispiel 5.4 illustriert diesen Zusammenhang.

Beispiel 5.4 (Spezifikations- und Komponentenzustände)

In diesem Beispiel zeigen wir den Unterschied zwischen Komponenten- und Automatenzuständen. Sei eine Komponente *konto* gegeben, mit der Zustandsmenge

$\text{state}_{\text{konto}} = (\{\text{bilanz, kreditrahmen}\} \rightarrow \text{Int})$

und der Eingabemenge

einzahlung(x), auszahlung(x), $x \in \mathbb{N}$.

Seien weiterhin die Automatenzustände *kreditwürdig*, *¬kreditwürdig* charakterisiert durch das Prädikat

$\text{kreditwürdig} : \text{state}_{\text{konto}} \rightarrow \mathbb{B}$

mit

$\text{kreditwürdig}(\sigma) \Leftrightarrow \sigma(\text{bilanz}) \geq \sigma(\text{kreditrahmen})$.

Das Automatendiagramm aus Abbildung 5.7 legt die Transitionsrelation fest, wobei jeder Pfeil eine Transition darstellt. Dabei geht der Pfeil vom Ausgangs- zum Zielzustand der Transition. Die Markierung des Pfeils gibt die Eingabe der Transition an.

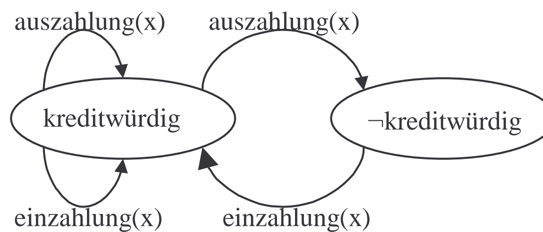


Abbildung 5.7: Automatendiagramm der Komponente *konto*.

□

Aus jedem Automaten A, dessen Zustände Klassen von Komponentenzuständen sind, kann auf kanonische Weise ein Automat A´ über Komponentenzuständen abgeleitet werden, der dasselbe Interaktionsverhalten „generiert. Deshalb und da wir auf Ebene der Entwicklungsprodukte von notationellen Aspekten abstrahieren, und damit von Aspekten der endlichen Darstellung von Automaten, unterscheiden wir nicht zwischen Spezifikations- und Komponentenzuständen.

Interpretation partieller Automaten

Neben den Automatenzuständen ist die Transitionsfunktion eines Automaten ein zentrales Element. Im Systemmodell lassen wir nur totale Automaten zu, um so zu garantieren, dass Komponenten ein nichtblockierendes Verhalten besitzen, das heißt *input enabled* sind (vergleiche Abschnitt 2.5.5). Dennoch muss es möglich sein, durch ein Entwicklungsprodukt nur bestimmte Eigenschaften einer Transitionsfunktion festzulegen. Diese Möglichkeit zu haben, ist methodisch wichtig, da nur so eine Entwicklung nach dem Prinzip der schrittweisen Verfeinerung beziehungsweise Vervollständigung möglich ist. Zudem erlaubt dies die strukturierte Beschreibung von Komponentenautomaten durch Entwicklungsprodukte. Ein einzelnes Produkt kann sich zum Beispiel auf die Transitionen beschränken, durch die

eine bestimmte (Art von) Eingabe, etwa die Aufrufe eines bestimmten Dienstes, verarbeitet werden.

In einer Spezifikation nur bestimmte Transitionen anzugeben bedeutet, dass für die Transitionsfunktion δ der Spezifikation gilt:

$$\exists \sigma, i. \delta(\sigma, i) = \varnothing .$$

Da wir im Systemmodell nur totale Automaten zulassen, stellt sich die Frage, wie wir den partiellen Automaten einer Spezifikation semantisch interpretieren. Hierfür gibt es verschiedene Möglichkeiten (vergleiche zum Beispiel [Rum96]):

- Unterspezifikation (lose Semantik).
  Für die „offenen Situationen" der Transitionsfunktion einer Spezifikation, das heißt für die Zustands-Eingabe-Paare, für welche die Transitionsfunktion die leere Menge liefert, werden keine Anforderungen an einen die Spezifikation erfüllenden Automaten gestellt.
- Wiederaufsetzen.
  In einer offenen Situation kann in einen beliebigen Zustand mit beliebiger Ausgabe übergegangen werden.
- Chaos.
  Mit dem (ersten) Erreichen einer offenen Situation ist beliebiges Verhalten möglich. Im Unterschied zum Wiederaufsetzen lässt die Chaos-Interpretation nicht nur für einen Schritt beliebiges Verhalten zu, sondern für alle auf eine offene Situation folgenden Schritte. Technisch bedeutet dies, dass ein zusätzlicher Chaoszustand eingeführt wird, von dem aus beliebige Transitionen möglich sind und in den mit beliebiger Ausgabe in einer offenen Situation gewechselt wird (vergleiche [Rum96]).
- Fehlerbehandlung.
  In einer offenen Situation wird in einen Fehlerzustand mit einer Fehlerausgabe übergegangen.
- Ignorieren.
  In einer offenen Situation wird die Eingabe ignoriert, das heißt, es erfolgt keine Ausgabe und keine Zustandsänderung.

Zu beachten ist, dass durch die lose Semantik einer Spezifikation eine *Menge* von Automaten zugeordnet wird, während alle anderen Interpretationsformen den partiellen Automaten der Spezifikation auf eindeutige Weise zu einem totalen Automaten vervollständigen/abschließen und somit einer Spezifikation genau ein Automat als Semantik zugeordnet wird.

Die Interpretation im Sinne der Unterspezifikation ist für die schrittweise Entwicklung und für eine kompositionale, ausschnittsweise Spezifikation von Komponentenautomaten relevant.

Der Abschluss durch Wiederaufsetzen und Chaos ist für Vorgehensweisen geeignet, in denen die Verfeinerung von Komponentenverhalten (durch Einschränkung von Nichtdeterminismus) im Vordergrund steht.

Der ignorierende Abschluss ist für Komponenten sinnvoll, für deren Verhalten nur bestimmte Eingaben von Bedeutung sind, zum Beispiel für filternde Komponenten.

Der Abschluss durch Fehlerbehandlung eignet sich für detaillierte, implementierungsnahe Spezifikationen.

Da wir die lose Semantik für ein modulares Vorgehen für unabdingbar halten und alle anderen Interpretationsformen auf analoge Weise in Produktarten umgesetzt werden können, beschränken wir uns auf die Definition der Produktart mit der losen Semantik:

Definition 5.13 (Spezifikation eines Komponentenautomaten)

Sei N eine nichtleere Menge von Nachrichten. Eine Automatenspezifikation besteht aus

- einem Komponentenbezeichner k
- einer Menge S von Zuständen
- einer endlichen Menge I von typisierten Kanalbezeichnern für Eingabekanäle
- einer endlichen Menge O von typisierten Kanalbezeichnern für Ausgabekanäle
- einer Transitionsfunktion $\delta : \left(S \times \vec{I^*}\right) \rightarrow \wp\left(S \times \vec{O^*}\right)$ sowie
- einer nichtleeren Menge von Initialelementen $init \subseteq \left(S \times \vec{O^*}\right)$.

Die Semantik einer Automatenspezifikation definieren wir wie folgt:

$$\begin{aligned}[\![(k,S,I,O,\delta,init)_{stm}]\!] =_{def} \{ sys \in SYS \mid\, & k \in K_{sys} \wedge \\ & (inits_k)_{sys} = init \wedge \\ & \forall i \in \vec{I^*}, s \in S.(\delta((i,s)) \neq \varnothing) \Rightarrow \\ & ((\delta_k)_{sys}((i,s)) = \delta((i,s))) \}\end{aligned}$$

□

Sei $sys \in SYS$ ein System. In obiger Definition garantiert die Forderung

$$k \in K_{sys}$$

die Wohldefiniertheit der beiden anderen Konjunktionsglieder, denn falls

$$\neg(k \in K_{sys})$$

gilt, enthält *sys* keine Elemente $inits_k$ und δ_k.

Der Zweck der Spezifikation eines Komponentenautomaten ist die Festlegung der Automatensicht einer Komponente. Grundlegende Produktarten, wie die Automatenspezifikation, definieren wir so, dass sie möglichst unabhängig (voneinander) sind (separation of concerns). Daher charakterisieren wir mit einer Automatenspezifikation in erster Linie nur die Systemmodellelemente, die ausschließlich für den Automaten einer bestimmten Komponente relevant sind. Dies sind die Initialmenge

und die Transitionsfunktion. Jedoch impliziert die Charakterisierung dieser Elemente Eigenschaften anderer Systemmodellelemente. Konkret ergeben sich die folgenden, impliziten Anforderungen an die syntaktische Schnittstelle und die Zustandsmenge der jeweiligen Komponente:

Die Formel

$$\forall i \in \vec{I^*}, s \in S.(\delta((i,s)) \neq \varnothing) \Rightarrow \left((\delta_k)_{sys}((i,s)) = \delta((i,s))\right) \quad (*)$$

aus obiger Definition, ist nur dann erfüllbar, wenn gilt

$$I = (I_k)_{sys} \wedge O = (O_k)_{sys} \quad (1)$$

Somit stellt (1) eine durch (*) implizierte Anforderung dar. Weiterhin ist (*) nur dann erfüllbar, falls gilt

$$\forall \sigma, \sigma', i, o.((\sigma', o) \in \delta(\sigma, o)) \Rightarrow \left(\sigma \in (state_k)_{sys} \wedge \sigma' \in (state_k)_{sys}\right)$$

und

$$\forall \sigma, o.((\sigma, o) \in init) \Rightarrow \left(\sigma \in (state_k)_{sys}\right)$$

Das bedeutet, dass die Zustände aus S, die in einer Transition aus δ als Quell- oder Zielzustand vorkommen oder Initialzustände sind, auch in $(state_k)_{sys}$ enthalten sein müssen. (*) impliziert daher auch

$$S_\delta \subseteq (state_k)_{sys} \quad (2)$$

wobei

$$\begin{aligned} S_\delta =_{def} \{\sigma \in S \mid \exists i, o, \sigma'.\\ &\delta(\sigma, i) \neq \varnothing \vee \\ &(\sigma, o) \in \delta(\sigma', i) \vee \\ &(\sigma, o) \in init\} \end{aligned}$$

Auf analoge Weise ergeben sich Anforderungen an die Kanaltypisierung $ctype_{sys}$ eines Systems $sys \in SYS$:

Für alle $c \in I \cup O$ definieren wir die Menge der in der Transitionsfunktion δ einer Spezifikation verwendeten Nachrichten durch

$$\begin{aligned} N_c =_{def} \{n \in N \mid \exists i, o, \sigma, \sigma'.\\ &((\sigma', o) \in \delta(\sigma, i) \vee (\sigma', o) \in init) \wedge \\ &((in(n, i(c))) \vee (in(n, o(c))))\} \end{aligned}$$

Damit ist (*) nur dann erfüllbar, falls für alle $c \in I \cup O$ gilt:

$$N_c \subseteq ctype_{sys}(c). \quad (3)$$

Durch Hinzunahme von (1), (2) und (3) in Definition 5.13 bliebe daher die Semantik einer Automatenspezifikation unverändert, jedoch könnten implizite Anforderungen explizit gemacht werden. Zudem ergeben sich aus (1), (2) und (3) entsprechende Konsistenzkriterien für Automatenspezifikationsprodukte und Ent-

wicklungsprodukte, welche Eigenschaften der Zustandsmenge einer Komponente oder die syntaktische Komponentenschnittstelle festlegen.

Eine *eindeutige/exakte* Spezifikation des Automaten einer Komponente ist eine Komposition aus

- einer Spezifikation der syntaktischen Schnittstelle der Komponente, welche die Menge der Ein- und Ausgabekanäle sowie die Typisierung der Eingabekanäle exakt festlegt,
- einer exakten Spezifikation der Zustandsmenge und
- einer Transitionsfunktion δ mit $\forall s,i.\delta(s,i) \neq \emptyset$.

Zustandsmenge und Typisierung der Eingabekanäle müssen exakt festgelegt werden, da die Totalität eines Automaten immer relativ zu seiner Zustands- und Eingabemenge definiert ist.

Im folgenden Beispiel legen wir das Verhalten der Datenbankkomponente unseres Bibliothekssystems durch eine Automatenspezifikation fest:

Beispiel 5.5 (Automatenspezifikationen der Bibliotheksdatenbank)

In diesem Beispiel beziehen wir uns auf die in Beispiel 5.1 spezifizierte Schichtenarchitektur des Bibliothekssystems, gegeben durch das Spezifikationsnetzwerk snw_1. Dabei nehmen wir für die Komponente *BibDB* an, dass ihr Zustandsraum dem fachlichen Zustandsraum des Anwendungssystem, das heißt der Bibliothek, entspricht, so dass gelte

$$(\text{states}_{\text{BibDB}})_{\text{sys}} = (\text{FS}_{\text{sys}})_{\text{State}}$$

Weiter legen wir fest, dass damit *BibDB* den fachlichen Zustandsraum realisiert. Das heißt, für

$$((\text{hs}_1)_{\text{sys}})_{\text{State}} : ((\text{KS}_1)_{\text{sys}})_{\text{State}} \rightarrow (\text{FS}_{\text{sys}})_{\text{State}}$$

gelte für alle $s \in ((\text{KS}_1)_{\text{sys}})_{\text{State}}$:

$$((\text{hs}_1)_{\text{sys}})_{\text{State}}(s) = s(\text{BibDB}).$$

Da *BibDB* die (Realisierung der) fachlichen Daten kapselt, muss es durch Interaktion mit *BibDB* möglich sein, Zustandsänderungen anzustoßen und Zustandseigenschaften abzufragen. In einem System *sys* sind die zustandsbezogenen Ereignismarkierungen T_{sys} und Z_{sys} die aus fachlicher Sicht relevanten Zustandsänderungen und -beobachtungen. Daher verstehen wir hier jede der Markierungen als Nachricht, so dass gelte

$$T_{sys} \subseteq N_{sys} \wedge Z_{sys} \subseteq N_{sys}$$

was durch die beiden folgenden Produkte ausgedrückt werden kann:

$$(T_{sys})_{\subseteq N}, (Z_{sys})_{\subseteq N}$$

Diese Nachrichten seien die zulässigen Eingaben an die Datenbankkomponente, so dass gelte

$$\forall c \in (In_{BibDB})_{sys}.ctype_{sys}(c) = T_{sys} \cup N_{sys}.$$

Für das Verhalten von *BibDB* haben wir folgende Vorstellung: Der Empfang einer Nachricht, der oben beschriebenen Art, durch die Datenbankkomponente impliziert folgende Reaktionen:

- im Fall einer Zustandsbeobachtung wird eine Nachricht mit dem der Anfrage entsprechenden Wert bezüglich des Komponentenzustands ausgegeben. Der Komponentenzustand bleibt unverändert.
- im Fall einer Zustandsänderungsmarkierung vollzieht die Komponente einen entsprechenden Zustandsübergang.

Eine, als Reaktion auf eine Zustandsbeobachtung ausgegebene Nachricht modellieren wir als 2-Tupel, bestehend aus der Anfrage und dem zugehörigen Rückgabewert. Es gelte somit

$$Reply \subseteq N_{sys}$$

mit

$$Reply =_{def} \left\{ (f,v) \in Z_{sys} \times \bigcup_{s \in sorts((\Sigma_{fs})_{sys}) \setminus \{State\}} (FS_{sys})_s \;\middle|\; \forall s \in sorts((\Sigma_{fs})_{sys}) \setminus \{State\}.\; \Pi_2(type(f)) = s \Rightarrow v \in (FS_{sys})_s \right\}$$

und

$$\forall c \in (Out_{BibDB})_{sys}.ctype_{sys}(c) = Reply$$

Das oben informell beschriebene Verhalten von *BibDB* geben wir nun präzise mittels zweier Automatenspezifikationen an. In dem Entwicklungsprodukt $EP_{STM_BibDB_1}$ spezifizieren wir das Reaktionsverhalten für Eingaben, die nur eine Nachricht umfassen. Damit betrachten wir nur die einfachsten Fälle, in denen wir uns nicht um die Behandlung gleichzeitig (auf einem oder verschiedenen Kanälen) eintreffender Nachrichten kümmern müssen.

Mit einem zweiten Produkt, $EP_{STM_BibDB_2}$ genannt, spezifizieren wir, aufbauend auf den Transitionen für die einelementigen Eingaben, das Verhalten für Eingaben mit mehr als einer Nachricht. Dabei fordern wir, im Sinne der Sequentialisierbarkeit von ACID-Transaktionen (vergleiche zum Beispiel [Tan92]), dass zu jeder Transition *t* mit mehrelementiger Eingabe, eine Folge von Transitionen mit einelementigen Eingaben existieren muss, die, nach Abstraktion von Zeit/Takt,

- die Eingabe von t (modulo Zeit) verarbeitet und
- dabei die Ausgabe der Transition t (modulo Zeit) erzeugt sowie
- in den Zielzustand von t führt.

Die Automatenspezifikation für einelementige Eingaben betten wir in ein Entwicklungsprodukt ein, um so in der Automatenspezifikation Bezug auf die Trägermenge $(FS_{sys})_{State}$ der fachlichen Zustände eines Systems *sys* nehmen zu können. Dies ist notwendig, da wir die Zustandsmenge des Automaten im Entwicklungsprodukt nicht explizit angeben, sondern uns auf die Menge der fachlichen Zustände beziehen.

Wir definieren das Entwicklungsprodukt

$$EP_{STM_BibDB_1} =_{def} (p_1)_{pred}$$

mit

$$p_1(sys) \Leftrightarrow_{def} sys \in \left[\!\left[\left(BibDB, (FS_{sys})_{State}, I, O, \delta_1, init \right)_{stm} \right]\!\right]$$

für alle $sys \in SYS$, wobei

$$I =_{def} \{bsdb_1, bsdb_2, \ldots, bsdb_m\}$$
$$O =_{def} \{dbbs_1, dbbs_2, \ldots, dbbs_m\}$$

und

$$init =_{def} \{(init_state_{sys}, \Omega)\}$$

seien, mit $m \in \mathbb{N}$. Die Ein- und Ausgabekanäle aus I und O seien wie oben informell angegeben typisiert und für eine Kanalmenge C sei $\Omega \in \vec{C^*}$ das „leere Interaktionsmuster", mit $\Omega(c) =_{def} \varepsilon$ für alle $c \in C$.

Für die Definition der Transitionsabbildung δ_1 führen wir die Hilfsfunktion

$$single_msg : \vec{C^*} \to \mathbb{B}$$

ein (wobei C eine endliche Menge von Kanalbezeichnern sei, die über einer Nachrichtenmenge N typisiert sind). Durch die Abbildung *single_msg* charakterisieren wir Interaktionsmuster über C, die genau eine Nachricht enthalten:

$$\begin{aligned} single_msg(\varphi) \Leftrightarrow_{def} \; & \exists c \in C, n \in N. \\ & \varphi(c) = \langle n \rangle \wedge \\ & \forall c' \in C. c' \neq c \Rightarrow \varphi(c') = \varepsilon \end{aligned}$$

Wir legen die Transitionsabbildung δ_1 eindeutig fest, indem wir fordern:

$$\forall s \in (state_{BibDB})_{sys}, \varphi \in \vec{I^*}, \psi \in \vec{O^*}, f \in T_{sys} \cup Z_{sys}, i \in [1,2, ... , m].$$
$$single_msg(\varphi) \wedge \varphi(bsdb_i) = \langle f \rangle \Rightarrow$$
$$((f \in T_{sys} \Rightarrow \delta_1(s,\varphi) = \{(f(s),\Omega)\}) \wedge$$
$$f \in Z_{sys} \Rightarrow (\delta_1(s,\varphi) = \{(s,\psi)\} \wedge$$
$$single_msg(\psi) \wedge \psi(dbbs_i) = \langle (f, f(s)) \rangle))$$
$$\wedge$$
$$\neg single_msg(\varphi) \Rightarrow \delta_1(s,\varphi) = \varnothing$$
$$\wedge$$
$$(\varphi = \Omega) \Rightarrow \delta_1(s,\varphi) = \{(s,\Omega)\}$$

Mit dem Produkt $EP_{STM_BibDB_1}$ machen wir keine Festlegungen bezüglich des Verhaltens von *BibDB* bei mehrelementigen Eingaben, da wir in obiger Definition von δ_1 festlegen, dass

$$\neg single_msg(\varphi) \Rightarrow \delta_1(s,\varphi) = \varnothing$$

gilt, und somit im Fall

$$\neg single_msg(\varphi)$$

gemäss Definition 5.13 mit $EP_{STM_BibDB_1}$ keine Festlegung von

$$(\delta_{BibDB})_{sys}(s,\varphi)$$

(für beliebigen Zustand s) gemacht wird.

Abbildung 5.8 illustriert, in Form einer Tabelle, die mit $EP_{STM_BibDB_1}$ festgelegten Eigenschaften der Transitionsfunktion. Diese Darstellung macht deutlich, dass für die Komponente BibDB Automatendiagramme, in denen Spezifikationszustände Äquivalenzklassen von Komponentenzuständen entsprechen, keine adäquate Darstellungsform sind, da im Falle von BibDB Äquivalenzklassen von Eingaben und nicht von Komponentenzuständen bedeutsam sind.

Eingabe $\varphi \in \vec{I}^*$	**Ausgangszustand**	**Zielzustand**	**Ausgabe** $\psi \in \vec{O}^*$
$single_msg(\varphi) \wedge \varphi(bsdb_i) = \langle f \rangle, f \in T_{sys}$	s	f(s)	Ω
$single_msg(\varphi) \wedge \varphi(bsdb_i) = \langle f \rangle, f \in Z_{sys}$	s	s	$\psi(dbbs_i) = \langle (f, f(s)) \rangle$

Abbildung 5.8: Tabellarische Darstellung von Eigenschaften der Transitionsfunktion δ_1

Mit dem unten angegebenen Produkt $EP_{STM_BibDB_2}$, vervollständigen wir die Verhaltensbeschreibung, indem wir die Reaktionen auf mehrelementige Eingaben festlegen. Wie erwähnt, legen wir diese Reaktionen fest, indem wir fordern, dass eine, modulo Zeit/Takt, entsprechende Folge von Reaktionen auf die einzelnen enthaltenen Eingaben existiert.

Für die Spezifikation der unten gegebenen Transitionsabbildung δ_2 benötigen wir eine Hilfsfunktion

$$flatten : (M^*)^\omega \to M^\omega$$

durch die wir von der „Taktung“ eines Stroms von Sequenzen über Elementen einer Menge M abstrahieren. Für alle $x \in M^*, xs \in (M^*)^\omega$ gelte:

$$flatten(\varepsilon) =_{def} \varepsilon$$
$$flatten(\langle x \rangle) =_{def} x$$
$$flatten(x \,\&\, xs) =_{def} x \circ flatten(xs)$$

Das Produkt $EP_{STM_BibDB_2}$ definieren wir wie folgt:

$$EP_{STM_BibDB_2} =_{def} (p_2)_{pred}$$

mit

$$p_2(sys) \Leftrightarrow_{def} sys \in \left[\!\left[\left(BibDB, (FS_{sys})_{State}, I, O, \delta_2, init \right)_{stm} \right]\!\right]$$

für alle $sys \in SYS$. Die Elemente I, O und init seien dabei wie oben definiert. Die Transitionsabbildung δ_2 legen wir, unter Verwendung der oben definierten Abbildung δ_1, eindeutig fest, indem wir fordern:

$$\forall s \in (\text{state}_{\text{BibDB}})_{\text{sys}}, \varphi \in \vec{I}^*, \psi \in \vec{O}^*.$$
$$(\text{single_msg}(\varphi) \vee (\varphi = \Omega)) \Rightarrow \delta_2(s,\varphi) = \delta_1(s,\varphi)$$
$$\wedge$$
$$\neg\text{single_msg}(\varphi) \Rightarrow$$
$$\delta_2(s,\varphi) = \Big\{(s',\psi) \in \Big((\text{state}_{\text{BibDB}})_{\text{sys}} \times \vec{O}^*\Big) \Big| \exists \xi \in \Big(IS_{(I \cup O),(\text{state}_{\text{BibDB}})_{\text{sys}}}\Big)^*.$$
$$s_1 = s \wedge$$
$$s_{\#\xi} = s' \wedge$$
$$\text{flatten}\big(\Pi_1(\xi)|_I\big) = \varphi \wedge$$
$$\text{flatten}\big(\Pi_1(\xi)|_O\big) = \psi \wedge$$
$$(\forall i \in \mathbb{N}.1 \le i < \#\xi \Rightarrow$$
$$(s_{i+1}, \text{output}_{i+1}) \in \delta_1(s_i, \text{input}_i))\}$$

wobei für alle $1 \le j \le \#\xi$ gelte

$$s_j =_{\text{def}} \Pi_2(\xi @ j)$$
$$\text{input}_j =_{\text{def}} \Pi_1(\xi @ j)|_I$$
$$\text{output}_j =_{\text{def}} \Pi_1(\xi @ j)|_O$$

□

Aus methodischer Sicht ist erwähnenswert, dass wir durch einen Automaten Sicherheitseigenschaften des Interaktionsverhaltens einer Komponente festlegen. Dadurch können wir zum Beispiel anhand des Automaten einer Komponente überprüfen, ob ein bestimmtes, beobachtbares Interaktionsverhalten Rückschlüsse auf den Komponentenzustand zulässt. Dies ist für den Zusammenhang zwischen zustandsbezogenen Prozessereignissen und deren Realisierung durch Komponenteninteraktion von Bedeutung. Betrachten wir hierzu als Beispiel die Anforderungen an die Realisierung der Ereignismarkierung

m.eqstatus(entliehen)

aus Beispiel 5.7. Aus obiger Automatenspezifikation folgt, dass, wenn auf eine Eingabe von

m.eqstatus(verfügbar)

zum Zeitpunkt j, die Ausgabe

(m.eqstatus(verfügbar), true)

(zum Zeitpunkt j+1) folgt, nicht unbedingt für den Zustand s zum Zeitpunkt j oder j+1

m.eqstatus(verfügbar)(s) gegolten haben muss.

Die folgende exemplarische Eingabe macht dies deutlich:

Eingabe zum Zeitpunkt j: <m.rückgabe(); m.eqstatus*(verfügbar); m.ausleihe*(l)>.

Damit trägt BibDB mit einem Verhalten gemäss obiger Automatenspezifikation nur im Falle bezüglich konfliktfreier Eingabe-Interaktionsmuster zu einer Realisierung der Ereignismarkierung m.eqstatus(verfügbar) bei.

5.4 Interaktionsverhalten einer Komponente

In obiger Automatensicht einer Komponente wird das Interaktionsverhalten mit Bezug auf die Zustände der Komponente charakterisiert. In manchen Fällen ist es jedoch sinnvoll, Eigenschaften des Interaktionsverhaltens einer Komponente ohne Bezugnahme auf Zustandsaspekte zu formulieren, beispielsweise im Falle von Komponenten, die keine fachlichen Daten realisieren und reine Interaktionsaufgaben übernehmen.

Für die zustandsunabhängige Verhaltensspezifikation führen wir die Produktart der *Spezifikation des Interaktionsverhaltens einer Komponente* ein.

Definition 5.14 (Spezifikation des Interaktionsverhaltens einer Komponente)

Eine Spezifikation des Interaktionsverhaltens besteht aus

- einem Komponentenidentifikator $k \in KID$
- zwei endlichen Mengen $I, O \subseteq CH$ von Kanalbezeichnern, sowie
- einem Prädikat $R : \overrightarrow{(I \cup O)} \rightarrow \mathbb{B}$.

Das Prädikat R legt die Verhaltensfunktion der Komponente k fest:

$$\begin{aligned} [\![(k, I, O, R)_{K_behav}]\!] =_{def} \{ sys \in SYS | \; & k \in K_{sys} \wedge \\ & (I_k)_{sys} = I \wedge (O_k)_{sys} = O \wedge \\ & \forall i \in \vec{I}. (F_k)_{sys}(i) = \left\{ o \in \vec{O} \middle| \exists \varphi \in \overrightarrow{(I \cup O)}. \right. \\ & \qquad \left. \varphi|_I = i \wedge \varphi|_O = o \wedge R(\varphi) \right\} \} \end{aligned}$$

□

In [BS00] werden unterschiedliche Spezifikationsstile, zum Beispiel *equational*- und *assume/guarantee*-Spezifikationen, vorgeschlagen. Ein Spezifikationsstil bestimmt dabei die Form und Struktur, in der das Prädikat *R* einer Verhaltensspezifikation beschrieben wird. Mit obiger Produktart lassen wir beliebige Spezifikationsstile zu.

5.5 Realisierungsbeziehung

Im Rahmen einer Entwicklung sind einerseits die zu realisierende fachliche Aufgabenstellung und andererseits das (zu entwickelnde) Softwaresystem und dessen Umgebung zu beschreiben. Darüber hinaus ist es aus mehreren Gründen sinnvoll, zu spezifizieren, auf welche Weise eine fachliche Aufgabenstellung durch das aus Softwaresystem und dessen Umgebung gebildete Netzwerk realisiert wird. Die Realisierungsbeziehung zwischen Aufgabenstellung und dem Software-Umgebungs-Netzwerk umfasst zum einen

- die Realisierung fachlicher Information/Daten, das heißt eines fachlichen Zustandsraums, durch die Information, welche die einzelnen Komponenten von Softwaresystem und Umgebung tragen, das heißt durch die Zustandsräume der Komponenten, und zum anderen
- die Realisierung fachlicher Abläufe durch Kooperation, das heißt Interaktion, der Software- und Umgebungskomponenten.

Grundsätzlich ermöglicht eine klare Dokumentation der Realisierungsbeziehung die Verfolgbarkeit fachlicher Anforderungen und verbessert dadurch die Änderbarkeit und Wartbarkeit eines Softwaresystems, da transparent wird, auf welche Komponenten und auf welche Weise sich Änderungen der fachlichen Aufgabenstellung auswirken. So fordert beispielsweise das V-Modell [BDI97] mit der sogenannten *Anforderungszuordnung* innerhalb einer *Systemarchitektur*, die Zuordnung von Anwenderanforderungen zu den Architekturelementen, in denen diese adressiert werden.

In diesem Abschnitt befassen wir uns zunächst mit der Spezifikation der Realisierung des fachlichen Zustandsraums und anschließend daran mit der Realisierung fachlicher Abläufe. Die betroffenen Systemmodell-Elemente sind in Abschnitt 2.6 unter dem Begriff der Perspektiven-Zusammenhänge zusammengefasst.

5.5.1 Zustandsrealisierung

In unserem Systemmodell erfassen wir die zu realisierende fachliche Information durch den fachlichen Zustandsraum, repräsentiert durch die Trägermenge der Sorte *State* der Zustandsalgebra *FS*. Der Zustandsraum, den die Komponenten des Softwaresystems und dessen Umgebung bilden, erfassen wir im Systemmodell durch die Trägermenge der Sorte *State* der jeweiligen Zustandsalgebra KS_i eines Spezifikationsnetzwerkes snw_i.

Auf welche Weise Netzwerkzustände fachliche Zustände repräsentieren/realisieren, beschreiben wir durch eine Abbildung von der Menge der Netzwerkzustände in die Menge der fachlichen Zustände. Für jedes Spezifikationsnetzwerk snw_i ist dies die Abbildung $(\mathrm{hs_i})_{\mathrm{State}}$, die Element des Rechenstruktur-Homomorphismus hs_i ist. Um sicherzustellen, dass diese Abbildung „sinnvoll“ ist, gibt ein Entwicklungssystem für jede Operation, die auf dem fachlichen Zustandsraum definiert ist, eine Operation auf dem Netzwerkzustandsraum an, so dass die beiden Zustandsräume bezüglich dieser Operationen „ähnlich“ sind. Im Systemmodell modellieren wir dies durch die Homomorphie von hs_i (vergleiche Definition 2.27).

Während sich der Netzwerkzustandsraum aus den Zustandsräumen der Komponenten des Netzwerkes ableitet, ergeben sich die Operationen auf dem Netzwerkzustandsraum (die „Entsprechungen“ der fachlichen Operationen sind) nicht aus Komponenteneigenschaften, sondern müssen zusätzlich angegeben werden. Im Systemmodell sind dies Operationen der Zustandsrechenstruktur KS_i des jeweiligen Spezifikationsnetzwerkes snw_i. Ein Teil der Charakterisierung der Zustandsrealisierung ist somit die Spezifikation der Zustandsrechenstruktur (KS_i) des jeweiligen Spezifikationsnetzwerkes (snw_i).

Häufig lassen sich Eigenschaften der Zustandsrechenstruktur eines Netzwerkes in Abhängigkeit zu der fachlichen Zustandsrechenstruktur formulieren (vergleiche etwa Beispiel 5.5, wo sich jede Operation der Netzwerkrechenstruktur durch Komposition der entsprechenden fachlichen Operation mit einer Projektion auf den Zustandsraum einer Komponente, welche den fachlichen Zustandsraum vollständig realisiert, ergibt). Diese Form von Zusammenhängen zwischen fachlicher und Netzwerkrechenstruktur lassen sich im allgemeinen am klarsten durch entsprechende Axiome (über der Menge der Entwicklungssysteme) ausdrücken (vergleiche Beispiel 5.5), so dass wir für diesen Zweck keine spezifischen Produktarten vorgeben. Für die Fälle, in denen die Zustandsrechenstrukturen für sich betrachtet, und etwa durch algebraische Spezifikation charakterisiert werden, ergeben sich Produktart analog zu denjenigen, die wir in Kapitel 3 zur Spezifikation der Entitätsrechenstruktur eines Entwicklungssystems eingeführen. Anstelle der Rechenstruktur-Klasse C_f, wie im Fall der Entitätsrechenstruktur-Spezifikation, würden sich die Produktarten zur Charakterisierung der Netzwerk-Rechenstrukturen auf die Rechenstruktur-Klassen C_i, $i \in [1,...,n_{snw}]$, beziehen.

5.5.2 Ereignisrealisierung

Neben dem fachlichen Zustandsraum umfasst eine fachliche Aufgabenstellung auch die fachlichen Abläufe, die durch Kooperation von Softwaresystem und dessen Umgebung zu realisieren sind. Bausteine von Abläufen sind markierte (Prozess-)Ereignisse, so dass sich die Realisierung von Abläufen aus den Realisierungen der Ereignisse ergibt.

Für die Spezifikation der Realisierung fachlicher Abläufe benötigen wir daher Produktarten, mit Hilfe derer wir angeben, welche Netzwerk-(Teil-)Verhalten korrekte Realisierungen der verschiedenen Arten von Ereignissen darstellen. An dieser Stelle fließen die Spezifika der unterschiedlichen Arten von Ereignismarkierungen ein. In Definition 2.11 haben wir die Menge der Markierungen aus Zustandsprädikaten, Zustandsänderungen und Aktionen aufgebaut. Die mit den jeweiligen Markierungsarten verbundenen, grundlegenden Anforderungen an die zur Realisierung geeigneten (Teil-)Ausführungsfolgen eines Netzwerkes, sind mit den in Definition 2.28 festgelegten Eigenschaften der Abbildungen hi_i im Systemmodell erfasst.

Eine Abbildung hi_i ordnet (für das jeweilige Spezifikationsnetzwerk snw_i) jeder Ereignismarkierung die Netzwerk-Verhalten zu, welche geeignete Realisierungen darstellen. Im allgemeinen gibt es mehrere Alternativen, um ein bestimmtes (fachliches) Prozessereignis zu realisieren (siehe Beispiel 5.6), so dass wir (durch die Abbildung hi_i) jeder Markierung eine *Menge* von (Teil-)Ausführungsfolgen zuordnen. Damit ergeben sich ähnliche Produktarten, wie wir sie zur Spezifikation von

Geschäftsprozessen in Abschnitt 4.1 einführen, wo zu jedem Auslöser eines Geschäftsprozesses die *Menge* der Reaktionsalternativen zu charakterisieren ist. Neben der exakten Angabe der Menge der Alternativen kann diese schrittweise sowohl „von unten“, als auch „von oben“ eingeschränkt werden. Erstere Form der Einschränkung erreichen wir durch die *kann-Spezifikation* genannte Produktart (synonym *Szenario*), letztere durch *muss-Spezifikationen*:

Definition 5.15 (Ereignisrealisierung)

Die Spezifikation einer Ereignisrealisierung bezieht sich immer auf ein bestimmtes Spezifikationsnetzwerk und besteht aus

- einem Index $i \in \mathbb{N}_+$ zur Festlegung des Spezifikationsnetzwerkes und
- einer Ereignismarkierung *m*, deren Realisierung charakterisiert wird durch
- eine Menge $X \subseteq \left(IS_{C_X,S_X}\right)^\omega$ von Strömen, wobei $C_X \subseteq CH$ und S_X eine nichtleere Zustandsmenge seien.

Wir unterscheiden drei Produktarten

- *kann-Spezifikation* $(i,m,X)_{\subseteq_hi}$ mit

$$[\![(i,m,X)_{\subseteq hi}]\!] =_{def} \{sys \in SYS \mid i \in \left[1,\ldots,(n_{snw})_{sys}\right] \wedge m \in M_{sys} \wedge X \subseteq (hi_i)_{sys}(m)\}$$

- *muss-Spezifikation* $(i,m,X)_{\supseteq_hi}$ mit

$$[\![(i,m,X)_{hi\subseteq}]\!] =_{def} \{sys \in SYS \mid i \in \left[1,\ldots,(n_{snw})_{sys}\right] \wedge m \in M_{sys} \wedge X \supseteq (hi_i)_{sys}(m)\}$$

- und die *exakte Spezifikation* $(i,m,X)_{hi}$ mit

$$[\![(i,m,X)_{hi}]\!] =_{def} \{sys \in SYS \mid i \in \left[1,\ldots,(n_{snw})_{sys}\right] \wedge m \in M_{sys} \wedge X = (hi_i)_{sys}(m)\}$$

□

In unserem Ansatz erfassen wir mit den Ereignisrealisierungen Systemaspekte, die im Unified Process [JBR99] in den sogenannten *Use Case Realizations* spezifiziert werden. In dem Vorgehensmodell aus [IBM97] finden wir zu diesem Zweck die sogenannten *Analysis Scenarios* und *Analysis Object Interaction Diagrams* sowie *Design Scenarios* und *Design Object Interaction Diagrams*. Im deutschen Standard

[BDI97] sind die Ereignisrealisierungen unter dem Abschnitt *Anforderungszuordnung* innerhalb einer *Systemarchitektur* einzuordnen.

Das folgende Beispiel illustriert die Rolle von Ereignisrealisierungsspezifikationen.

Beispiel 5.6 (muss-Spezifikation von Ereignisrealisierungen)

Wir beziehen uns auf die Netzwerksicht aus Beispiel 5.1, die durch das Spezifikationsnetzwerk snw_1 einer (hypothetischen) Entwicklung erfasst sei. In dieser Sicht ist das Bibliotheks-Softwaresystem durch das Entwicklungsnetzwerk *dnw* erfasst, die Rollen des Lesers und der Bibliotheksangestellten durch die Komponenten *Leser* beziehungsweise *BibAngest*.

Bezüglich dieser Architektur legen wir fest, dass die Äußerung eines Vormerkungswunsches das Senden einer entsprechenden Nachricht von einem Leser oder einer Bibliotheksangestellten an das Bibliothekssoftwaresystem umfasst.

In der in Beispiel 4.1 gegebenen Prozesssicht, haben wir die Äußerung eines Vormerkungswunsches für ein Medium *m* und einen Leser *l* durch die Ereignismarkierung

vormerkungswunsch(m,l)

modelliert. In dem folgenden Entwicklungsprodukt, das wir *VMREAL* nennen, legen wir fest, dass die zu sendende Nachricht gleich der zu realisierenden Ereignismarkierung sei. Hierzu spezifizieren wir die Vormerkungsmarkierungen als Teilmenge der Nachrichtenmenge, und geben für jede Markierung eine muss-Spezifikation der Ereignisrealisierung an. Die Sequenzdiagramme in Abbildung 5.9 illustrieren die Interaktionsmuster, welche diesen Spezifikationen zugrunde liegen.

$$\mathrm{VMREAL} =_{\mathrm{def}} (\mathrm{p})_{\mathrm{pred}}$$

mit

$$\begin{aligned}
\mathrm{p}(\mathrm{sys}) \Leftrightarrow_{\mathrm{def}}\ & \mathrm{sys} \in [\![(\Sigma_{\mathrm{ACT}}, \mathrm{C}_{\mathrm{ACT}})_{\mathrm{muss_red}}]\!] \wedge \\
& \forall \mathrm{m} \in (\mathrm{A}_{\mathrm{sys}})_{\mathrm{id}_{\mathrm{Medium}}}, \mathrm{l} \in (\mathrm{A}_{\mathrm{sys}})_{\mathrm{id}_{\mathrm{Leser}}} . \\
& \mathrm{sys} \in [\![\left(\left\{ \mathrm{vormerkungswunsch}^{\mathrm{A}_{\mathrm{sys}}}(\mathrm{m},\mathrm{l}) \right\}\right)_{\subseteq \mathrm{N}}]\!] \wedge \\
& \mathrm{sys} \in [\![\left(1, \mathrm{vormerkungswunsch}^{\mathrm{A}_{\mathrm{sys}}}(\mathrm{m},\mathrm{l}), \mathrm{vmreal}(\mathrm{m},\mathrm{l})\right)_{\mathrm{hi}\subseteq}]\!]
\end{aligned}$$

für alle $\mathrm{sys} \in \mathrm{SYS}$. Dabei seien die Signatur Σ_{ACT} und die Klasse $\mathrm{C}_{\mathrm{ACT}}$ von Σ_{ACT}-Algebren durch die, in Beispiel 4.1 gegebene Spezifikation ACT charakterisiert.

Zudem sei die Funktion

$$\mathrm{vmreal} : (\mathrm{A}_{\mathrm{sys}})_{\mathrm{id}_{\mathrm{Medium}}} \times (\mathrm{A}_{\mathrm{sys}})_{\mathrm{id}_{\mathrm{Leser}}} \rightarrow \wp\left((\mathrm{IS}_1)_{\mathrm{sys}}^{\ *} \right)$$

für alle $\mathrm{m} \in (\mathrm{A}_{\mathrm{sys}})_{\mathrm{id}_{\mathrm{Medium}}}, \mathrm{l} \in (\mathrm{A}_{\mathrm{sys}})_{\mathrm{id}_{\mathrm{Leser}}}$ definiert durch

$$\begin{aligned}\mathrm{vmreal}(m,l) =_{def} \{&\varphi \in (IS_l)_{sys}^{*} \mid \\ &\exists s \in \mathrm{states}((snw_1)_{sys}), \psi \in \mathrm{channels}((snw_1)_{sys})^{\vec{*}}, j \in \mathbb{N}. \\ &(\mathrm{in}(\mathrm{vormerkungswunsch}^{A_{sys}}(m,l), \psi(lsw)) \vee \\ &\ \mathrm{in}(\mathrm{vormerkungswunsch}^{A_{sys}}(m,l), \psi(bsw))) \\ &\wedge \\ &\varphi @ j = (s, \psi)\}\end{aligned}$$

Mit der in [Krü01] gegebenen Semantik für Sequenzdiagramme, entspricht das mit *vw* bezeichnete Sequenzdiagramm aus Abbildung 5.9 der Repräsentation von *vmreak(m,l)*, für gegebenes *m* und *l*.

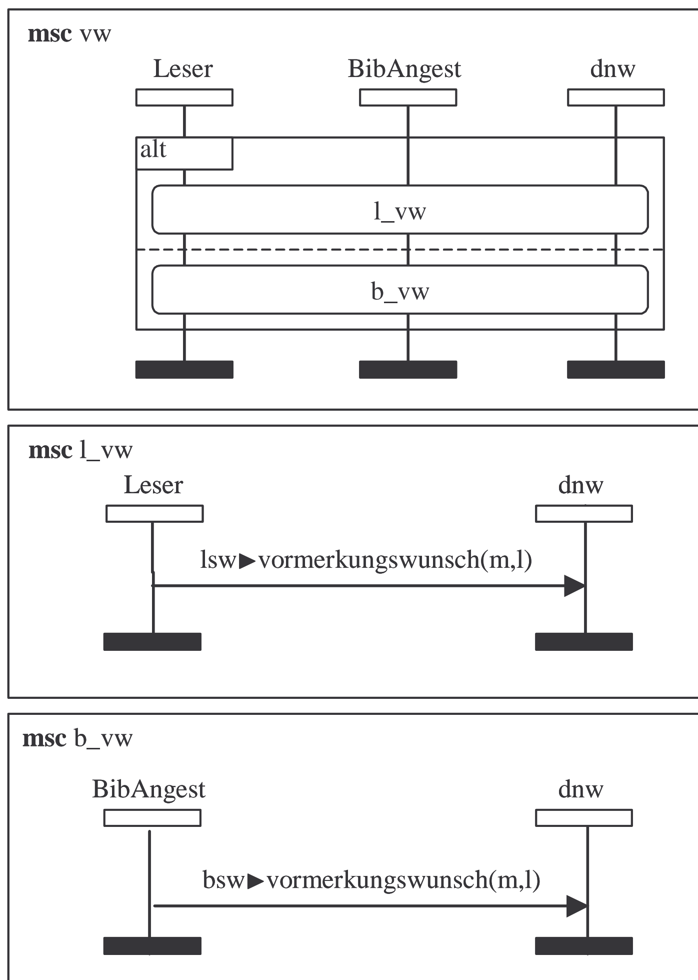


Abbildung 5.9: Interaktionsalternativen zur Realisierung des Prozessereignisses der Äußerung eines Vormerkungswunsches.

□

Das folgende Beispiel illustriert die Realisierung eines Zustandsbeobachtungs-Ereignisses.

Beispiel 5.7 (Realisierung einer Zustandsbeobachtung)

Das in Abbildung 5.10 gegebene Sequenzdiagramm stellt die Menge der Ausführungsfolgen des in Beispiel 5.2 charakterisierten Spezifikationsnetzwerkes snw_1 dar, welche wir als Realisierungen von Ereignissen verstehen, die mit dem Zustandsprädikat

m.eqstatus(verfügbar)

(vergleiche Beispiel 4.1)

markiert sind, betrachten. Ein mit *m.eqstatus(verfügbar)* markiertes Ereignis steht für die Prüfung des Statuswertes eines Mediums *m* auf Gleichheit mit dem Wert *verfügbar*. Diese Form der Zustandsbeobachtung ist Teil einer Vormerkungsbearbeitung (vergleiche Beispiel 4.1).

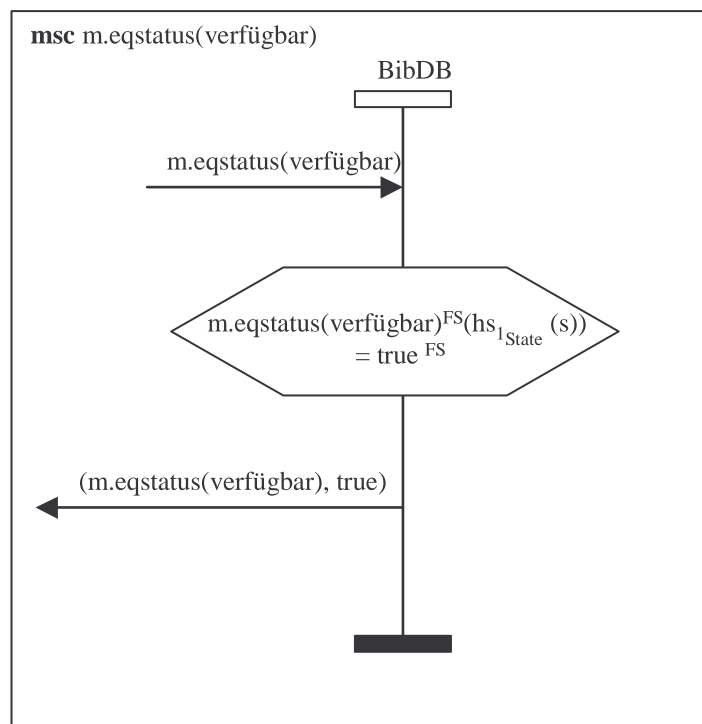


Abbildung 5.10: Ausführungsfolgen zur Realisierung einer Zustandsbeobachtung.

□

6 Komponierte Entwicklungsprodukte

In Abschnitt 2.2 führen wir Entwicklungsprodukte als Mittel zur (systematischen und strukturierten) Erfassung von Entwicklungsinhalten, wie etwa den Objekten der Anwendungswelt einer fachlichen Aufgabenstellung, ein. Die in den Kapiteln 3 bis 5 definierten (Basis-)Produktarten adressieren jeweils Entwicklungsinhalte, die sich im wesentlichen auf einen Aspekt eines Entwicklungssystems beziehen. Beispielsweise adressieren wir mit einer Invariantenspezifikation (gemäss Definition 4.5) nur den Aspekt der Sicherheitseigenschaften des (Anwendungssystem-)Verhaltens. Dementsprechend dienen diese (Basis-)Produktarten der Dokumentation von Ergebnissen einzelner Entwicklungsschritte, das heißt der Strukturierung von Entwicklungsinhalten im Kleinen.

Um eine Entwicklung im Grossen zu strukturieren und zu dokumentieren, benötigen wir umfassendere Produktarten, die zum einen mehrere, inhaltlich zusammengehörige Ergebnisse zu einer Einheit bündeln, und zum anderen diesen Ergebnissen eine zusätzliche Bedeutung geben, indem sie diese beispielsweise als in gewisser Hinsicht vollständig auszeichnen. Zum Beispiel können wir eine Verhaltensspezifikation, die sowohl Lebendigkeits- als auch Sicherheitseigenschaften festlegt, als die detaillierteste/feinste Spezifikation von Verhaltenseigenschaften im Rahmen einer Systementwicklung interpretieren, und damit ausschließen, dass andere Produkte dieser Systementwicklung darüber hinausgehende Verhaltenseigenschaften festlegen.

Umfassende Produktarten stellen häufig das Ergebnis einer bestimmten Projektphase dar und sind in vielen Fällen notwendige Grundlage für daran anschließende Phasen. Beispielsweise ist in einer Entwicklung nach dem Wasserfall-Modell das Vorliegen einer (aus fachlicher Sicht) vollständigen Spezifikation der fachlichen Anforderungen an das zu entwickelnde Softwaresystem, Voraussetzung für die Entwurfsphase (vergleiche etwa [Bal96]).

Selbst wenn innerhalb des Verlaufs einer Entwicklung Vollständigkeitsannahmen häufig mehr Wunschdenken als Realität sind, die Notwendigkeit von Rückkopplungsschleifen in (evolutionären) Entwicklungsprozessen deutet darauf hin, so dienen Produktarten, die einen Vollständigkeitsanspruch ausdrücken, zumindest dazu, diesen Anspruch, das heißt den intendierten Status eines Produktes, explizit zu machen. Nur dadurch wird etwa ein sinnvolles Änderungs-Management ermöglicht. Wird beispielsweise eine Verhaltensspezifikation des Anwendungssystems als die feinste Verhaltensspezifikation ausgezeichnet, so ist bei jeder neu hinzukommenden Verhaltensspezifikation zu prüfen, ob diese durch die feinste Spezifikation impliziert wird, und etwa nur eine in anderer Form gegebene aber inhaltlich

redundante Darstellung ist, oder ob die feinste Spezifikation entsprechend zu verstärken ist.

Neben der Bedeutung für das Änderungs-Management innerhalb einer Entwicklung, hat die Auszeichnung von Produkten mit einem gewissen Vollständigkeitsanspruch auch Bedeutung für, sich an eine Entwicklung anschließende Wartung und das Reengineering von Softwaresystemen, da diese ausgezeichneten Produkte hierfür geeignete Referenzdokumente darstellen. Beispielsweise stellt *die* Anforderungsspezifikation einer Entwicklung (vergleiche Definition 6.7) das Referenzdokument bezüglich der fachlichen Anforderungen an ein Softwaresystem dar.

In den folgenden Abschnitten bauen wir unter Verwendung der (Basis-)Produktarten aus Kapitel 3 bis 5 umfassende und in gewisser Hinsicht vollständige Produktarten auf.

Die vollständige Spezifikation einer fachlichen Aufgabenstellung ist Gegenstand von Abschnitt 6.1. Anschließend daran, gehen wir auf die Spezifikation der Grenze eines zu entwickelnden Softwaresystems ein, wobei wir die Systemgrenze zum einen durch Verteilung fachlicher Aufgaben auf Softwaresystem und -Umgebung angeben. Damit ergibt sich die fachliche Sicht auf Zustands- und Verhaltenseigenschaften des zu entwickelnden Softwaresystems. Ein zweiter Aspekt der Softwaresystemgrenze ist die Nutzungssicht, in der wir die Interaktionsformen zwischen Softwaresystem und Umgebung angeben, durch welche fachliche Aufgaben in Kooperation von Softwaresystem und Umgebung realisiert werden.

Sowohl die Spezifikation der fachlichen Aufgabenstellung als auch der Softwaresystemgrenze sehen wir als Bestandteile (der Produktart) der Anforderungsspezifikation, die wir in Abschnitt 6.3 behandeln.

Da wir die Realisierung des zu entwickelnden Softwaresystems in Form eines Komponentennetzwerkes, *dnw* genannt (vergleiche Definition 2.16), modellieren, eignen sich Komponentennetzwerke auch zur *Spezifikation* von Eigenschaften der Realisierung. Mit den sogenannten Spezifikationsnetzwerken des Systemmodells aus Abschnitt 2.5.4, erfassen wir die in Netzwerkform gegebenen, spezifikatorischen Sichten auf eine Realisierung. In Abschnitt 6.4 führen wir die Produktart der Architektur ein, die zur vollständigen Angabe einer solchen spezifikatorischen Sicht dient. Wir unterscheiden dabei insbesondere die

- (feinste) Schnittstellenarchitektur des Softwaresystems, mit Hilfe derer wir, durch Festlegung aller einzuhaltenden Schnittstellen einer Realisierung, Anforderungen an die Modularisierung, das heißt an die Topologie, des zu entwickelnden Softwaresystems formulieren
- (feinste) Realisierungsarchitektur des Softwaresystems, welche die Topologie des Softwaresystems eindeutig bestimmt, und die stärksten Anforderungen an das beobachtbare Verhalten der Software-Komponenten enthält
- (feinste) Umgebungsarchitektur, um die (zulässigen Annahmen über) die Eigenschaften der Umgebung des Softwaresystems angeben zu können.

6.1 Spezifikation der fachlichen Aufgabenstellung

Die Beschreibung der fachlichen Aufgabenstellung, die mit Hilfe des zu entwickelnden Softwaresystems zu realisieren ist, hat sowohl im Forward- als auch im Reverse-Engineering seine Bedeutung. Im Forward-Engineering dient sie als Ausgangspunkt für die Festlegung der Rolle, die das Softwaresystem im Anwendungskontext einnehmen soll. Gegebenenfalls werden mehrere, alternative Rollen des Softwaresystems einander gegenüber gestellt, um dann eine dieser Alternativen als Grundlage für die weitere Entwicklung auszuwählen. Ein solcher Auswahlprozess ist typisch für die Aufgabe des Requirements-Engineering, die häufig in die drei Teilaufgaben

- *Elicitation*
- *Negotiation* und
- *Decision*

gegliedert wird (vergleiche beispielsweise [Poh96]).

Im Reverse-Engineering dient die Beschreibung der fachlichen Aufgabenstellung vornehmlich einem verbesserten Verständnis der fachlichen Aufgaben des Softwaresystems, indem der fachliche Kontext in seiner Gesamtheit beschrieben wird, und dies in einer Form, die unabhängig ist, von der zugrundeliegenden Komponentenstruktur.

Üblicherweise entsteht die Beschreibung der fachlichen Aufgabenstellung in vielen kleinen Entwicklungsschritten, die jeweils durch entsprechende (kleine) Entwicklungsprodukte dokumentiert werden. Für die Systementwicklung ist es jedoch von Bedeutung, ein Entwicklungsprodukt als die *vollständige* Beschreibung der fachlichen Aufgabenstellung auszeichnen zu können. Zweck eines solchen Produkts ist es, als *die* Referenz für alle fachlichen Fragestellungen, die im Rahmen einer Systementwicklung auftreten, zu dienen. Wenn beispielsweise zu klären ist, aus welchen Schritten ein bestimmter Geschäftsprozess, etwa die Vormerkung eines Mediums einer Bibliothek, aufgebaut ist, so muss sichergestellt sein, dass die in dem ausgezeichneten Produkt gegebene Geschäftsprozessspezifikation, die „feinste" Spezifikation des Prozesses ist, welche die Produktmenge der betrachteten Entwicklung umfasst.

Das Charakteristikum, die vollständige Beschreibung der fachlichen Aufgabenstellung zu sein, weisen wir der, im folgenden eingeführten, Produktart der *Spezifikation der fachlichen Aufgabenstellung* zu. Diese Art von Entwicklungsprodukt umfasst drei Teilsichten:

- die *Datensicht*, welche die Sorten fachlicher Entitäten des Systems und ihre charakteristischen Operationen festlegt.
- die *fachliche Zustandssicht*, welche die Menge der Entitäten festlegt, die den Zustandsraum der ganzheitlichen Perspektive aufspannen, sowie den fachlichen Initialzustand.
- die *Prozesssicht*, welche die Menge der (fachlich) zulässigen Prozesse definiert.

Indem wir von den drei Teilsichten fordern, dass diese die adressierten Entwicklungsinhalte auf eindeutige Weise festlegen, stellen wir sicher, dass eine ganzheitliche Sicht des Anwendungssystems eine *vollständige* Beschreibung dieser Entwicklungsinhalte, das heißt der entsprechenden Entwicklungssystem-Elemente, darstellt.

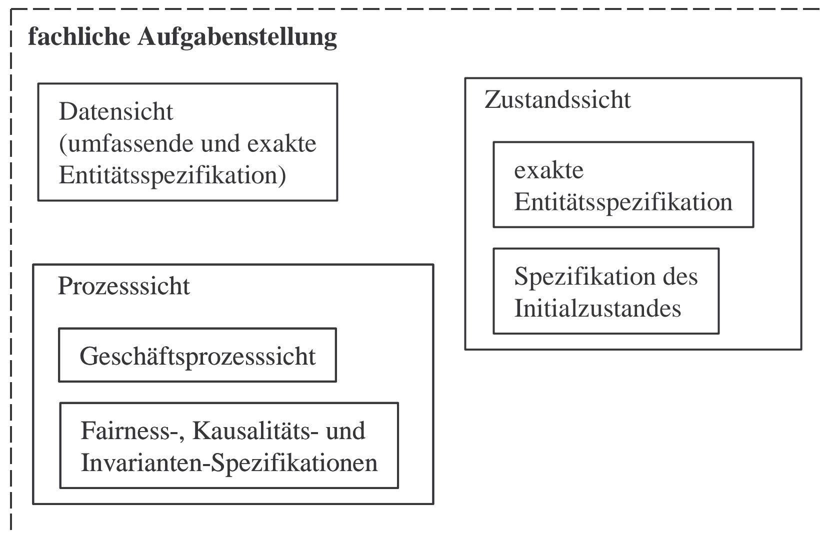


Abbildung 6.1: Produktarten zur Spezifikation einer fachlichen Aufgabenstellung.

Abbildung 6.1 zeigt die Produktarten zur Spezifikation der fachlichen Aufgabenstellung.

6.1.1 Fachliche Daten- und Zustandssicht

Zweck einer fachlichen Datensicht ist es, die Sorten fachlicher Werte/Information und Operationen darauf vollständig zu spezifizieren. Grob gesagt, legen wir mit einer fachlichen Datensicht die fachlichen, abstrakten Datentypen fest. In unserem Systemmodell erfassen wir die fachlichen Sorten und Operationen durch die Entitätssignatur Σ_{fe} und die Klasse C_f der fachlichen Entitätsrechenstrukturen (vergleiche Definition 2.6).

Den Zweck einer fachlichen Datensicht erfüllt die Produktart der *exakten Spezifikation der Entitätsrechenstrukturen* aus Definition 3.5, so dass wir den Begriff der fachlichen *Datensicht* hier lediglich als Synonym für die Produktart aus Definition 3.5 einführen.

Um anzugeben, wie sich der fachliche Zustandsraum (des betrachteten Anwendungssystems) zusammensetzt, müssen wir die Menge der Entitäten angeben, die den Zustandsraum aufspannen. Die Produktart der *exakten Entitätsspezifikation* aus Definition 3.6 dient dazu, die Entitätsmenge *E* eines Systems eindeutig festzulegen. Zusammen mit der Festlegung der Initialzustände der Entitäten gemäss Definition 3.7 erfassen wir die fachlichen Zustandsaspekte in der sogenannten *Zustandssicht* vollständig:

Definition 6.1 (Fachliche Zustandssicht)

Als *Zustandssicht* bezeichnen wir eine 2-elementige Produktmenge, die aus

- einer exakte Spezifikation der Entitätsmenge $(X)_E$ (siehe Definition 3.6), wobei X eine endliche Menge sei, und
- einer Spezifikation des Initialzustands $(s_0)_{init_state}$ (siehe Definition 3.7), wobei

  $s_0 : X \rightarrow VAL$

  und VAL eine nichtleere, abzählbare Menge sei,

besteht.

□

Beispiel 6.1 gibt die Zustandssicht unseres Bibliotheksbeispiels an:

Beispiel 6.1 (Daten- und Zustandssicht der Bibliothek)

Interpretieren wir die in Beispiel 4.1 gegebene Spezifikation ACT, deren Semantik durch die Signatur Σ_{ACT} und die Rechenstruktur-Klasse C_{ACT} gegeben sei, als umfassend und exakt, so stellt sie eine geeignete fachliche Datensicht für unser Bibliotheksbeispiel dar:

$$BibDataView =_{def} (\{Medium, Status, Leser\}, \Sigma_{ACT}, C_{ACT})_{fe}$$

Spezifikation ACT ist hierfür geeignet, da sie auf der Spezifikation MEDIUM´ aus Beispiel 3.2 aufbaut, mit der die Sorten und Operationen fachlicher Entitäten der Bibliothek gegeben sind, und zusätzlich die notwendigen Aktionsbezeichner angibt. Wie erwähnt, modellieren wir Aktionsbezeichner als Sorte der Entitätsrechenstruktur, da Aktionsbezeichner häufig abhängig von Entitäten sind, beispielsweise der Aktionsbezeichner

vormerkungswunsch(m,l)

mit den Entitätsbezeichnern *m* und *l* für das vorzumerkende Medium *m* und den Leser *l*.

Bezüglich der Zustandsaspekte der Bibliothek sei der Medienbestand durch die (endliche) Menge der (Bezeichner von) Medien-Entitäten gegeben, der Leserstamm durch die Menge der (Bezeichner von) Leser-Entitäten. Den Initialzustand wählen wir so, dass alle Medien- und Leserentitäten in einem jeweils einheitlichen Initialzustand sind, gegeben durch die (konstanten) Konstruktoroperationen

$mkons : \rightarrow Medium$

und

$lkons : \rightarrow Leser$

(vergleiche Spezifikation MEDIUM´ aus Beispiel 3.2). Wir gehen davon aus, dass *Leser* und *Medium* als Wertsorten spezifiziert seien, und definieren die Zustandssicht der Bibliothek wie folgt:

$$\text{BibStateView} =_{\text{def}} \left\{ \left((A_{sys})_{id_{Leser}} \cup (A_{sys})_{id_{Medium}} \right)_E, (s_0)_{init_state} \right\}$$

mit

$s_0(m) = \text{mkons}$

und

$s_0(l) = \text{lkons}$

für alle $m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}}$.

□

6.1.2 Prozesssicht

Teil einer vollständigen Spezifikation einer fachlichen Aufgabenstellung ist die Spezifikation *aller* fachlich relevanten Verhaltenseigenschaften des betrachteten Anwendungssystems.

In der ganzheitlichen Perspektive unseres Systemmodells bilden wir das Systemverhalten (des Anwendungssystems) durch die Menge *P* der Systemprozesse ab (vergleiche Definition 2.12). Da wir *P* als die Menge der zulässigen Systemverhalten verstehen (vergleiche Definition 2.30), bedeutet die Spezifikation *aller* Verhaltenseigenschaften die *eindeutige* Festlegung von *P*. Um diese Prozessmenge zu charakterisieren führen wir in Kapitel 4 die Produktarten zur Spezifikation von

- Reaktionseigenschaften/Geschäftsprozessen,
- Fairness- und Kausalitätseigenschaften, sowie
- Invarianten

ein. Dementsprechend umfasst die Produktart der Prozesssicht, die den Zweck einer vollständigen, fachlichen Verhaltensspezifikation hat, eine vollständige Menge der genannten Produktarten.

Als einen Bestandteil einer Prozesssicht fassen wir die Spezifikation aller Reaktionseigenschaften in der sogenannten *Geschäftsprozesssicht* zusammen. Eine Geschäftsprozesssicht umfasst für jeden, durch das Softwaresystem zu unterstützenden Geschäftsprozess, eine exakte Reaktionsspezifikation. Durch eine „Abschlussoperation“ in der Semantik einer Geschäftsprozesssicht schließen wir aus, dass weitere Reaktionseigenschaften gefordert werden:

Definition 6.2 (Geschäftsprozesssicht)

Eine Geschäftsprozesssicht besteht aus

- einer endlichen Menge von 2-Tupeln aus $M \times \wp([\text{pcs}_{\text{fin}}(\text{EV}, M)])$,

die wir mit

$(m_1, X_1), \ldots, (m_n, X_n),\ n \in \mathbb{N}$

bezeichnen.

Die Semantik definieren wir durch

$$\llbracket ((m_1, X_1), \ldots, (m_n, X_n))_{reak!} \rrbracket =_{def} \{sys \in SYS | \\ \{m_1, \ldots, m_n\} \subseteq M_{sys} \wedge \\ sys \in \llbracket \{(m_1, X_1)_{reak}, \ldots, (m_n, X_n)_{reak}\} \rrbracket \wedge \\ \forall m \in M_{sys} \setminus \{m_1, \ldots, m_n\}.reak_{sys}(m) = \{p_\varnothing\}\}$$

Da der leere Prozess $p_\varnothing$ Präfix eines jeden Prozesses ist, bedeutet für ein $m \in M_{sys}$ die Forderung

$reak_{sys}(m) = \{p_\varnothing\}$

keine (weitere) Einschränkung der Menge P_{sys} der zulässigen Prozesse eines Systems.

□

Im Systemmodell erfassen wir Reaktionseigenschaften durch die Abbildung *reak* (siehe Definition 2.13), welche Reaktionsauslöser (in Form einer Ereignismarkierung) mit der Menge der zugehörigen Reaktionsalternativen (in Form einer Menge von Prozess-Isomorphieklassen) in Bezug setzt. Die Vollständigkeit modellieren wir, indem wir die Produktsemantik so wählen, dass für alle Markierungen, die nicht als Auslöser in der Menge der Einzelspezifikationen auftreten, keine Reaktion gefordert wird. Damit schließen wir die, durch die Menge der Einzelspezifikationen gegebenen Anforderungen an die Abbildung *reak* ab, zu einer *eindeutigen* Charakterisierung von *reak*.

Durch das Charakteristikum der Vollständigkeit unterscheidet sich eine Geschäftsprozesssicht von einer Menge von Geschäftsprozessspezifikationen.

Darauf aufbauend ergibt sich eine Prozesssicht durch Hinzunahme von Fairness-, Kausalitäts- und Invariantenspezifikationen wie folgt:

Definition 6.3 (Prozesssicht)

Eine Prozesssicht besteht aus einer

- Geschäftsprozesssicht G (gemäss Definition 6.2), sowie
- einer endlichen Menge Q von Fairness-, Kausalitäts- und Invariantenspezifikationen (gemäss Definition 4.2 bis Definition 4.5)

Die Semantik definieren wir durch

$$
\begin{aligned}
[\![(G,Q)_P]\!] =_{def} \{ & sys \in SYS \mid \\
& sys \in [\![\{G,Q\}]\!] \wedge \\
& (\forall sys' \in SYS. \\
& sys' \in [\![\{G,Q\}]\!] \Rightarrow P_{sys'} \subseteq P_{sys}) \}
\end{aligned}
$$

Damit fordern wir, dass, außer den durch die Geschäftsprozesssicht G und die Spezifikationen Q gegebenen Prozesseigenschaften, keine weiteren Anforderungen an die Menge der (zulässigen) Prozesse gestellt werden, indem zu gelten hat, dass P_{sys} die größtmögliche Prozessmenge ist (ausgedrückt durch die Implikation der Semantikdefinition). Das bedeutet, dass eine Prozesssicht *alle* Anforderungen an die Menge der (zulässigen) Prozesse eines Systems enthält.

□

Mit Beispiel 6.2 geben wir die Prozesssicht für unser Bibliotheksbeispiel an:

Beispiel 6.2 (Prozesssicht der Bibliothek)

In unserem Bibliotheksbeispiel gehen wir davon aus, dass das zu entwickelnde Softwaresystem die Geschäftsprozesse der Vormerkung, Ausleihe und Rückgabe von Medien zu unterstützen hat. Dementsprechend umfasst eine Prozesssicht die exakte Spezifikation der drei genannten Formen von Geschäftsprozessen. Weiterhin werden jeweils die erwünschten Reaktionsalternativen durch Fairnessspezifikationen (siehe etwa Beispiel 4.4) ausgezeichnet. Kausalitäts- und Invariantenspezifikationen, wie etwa die aus Beispiel 4.4 und Beispiel 4.5, vervollständigen die Prozesssicht, die wir folgendermaßen angeben:

Seien *Medium* und *Leser* als Wertsorten spezifiziert und seien

$$
\begin{aligned}
& \text{vormerkungswunsch} : id_{Medium} \times id_{Leser} \rightarrow Act \\
& \text{ausleihe} : id_{Medium} \times id_{Leser} \rightarrow Act \\
& \text{rückgabe} : id_{Medium} \rightarrow Act
\end{aligned}
$$

Konstruktoren für Aktionsbezeichner (die wir als Operationen der Entitätssignatur Σ_{fe} modellieren).

Sei weiterhin für alle $m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}}$ durch

- XV(m,l) die Menge aller Reaktionsalternativen zu Auslöser $vormerkungswunsch^{A_{sys}}(m,l)$ (siehe Beispiel 4.3)
- FV(m,l), $FV(m,l) \subseteq XV(m,l)$ die Menge der fachlich erwünschten Reaktionsalternativen zu Auslöser $vormerkungswunsch^{A_{sys}}(m,l)$ (siehe Beispiel 4.4)
- XA(m,l), FA(m,l), $FA(m,l) \subseteq XA(m,l)$ die Menge aller sowie die Menge der fachlich erwünschten Reaktionsalternativen zu Auslöser $ausleihe^{A_{sys}}(m,l)$

- XR(m,l), FR(m), $FR(m) \subseteq XR(m)$ die Menge aller sowie die Menge der fachlich erwünschten Reaktionsalternativen zu Auslöser $rückgabe^{A_{sys}}(m)$

gegeben. Dann definieren wir die Geschäftsprozesssicht *BibGPView* unserer Bibliothek durch

$$BibGPView =_{def} (G)_{reak!}$$

wobei

$$\begin{aligned} G =_{def} \{(a,R) \in M_{sys} \times \wp([pcs_{fin}(EV, M_{sys})]) | \\ \exists m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}} . \\ (a = vormerkungswunsch^{A_{sys}}(m,l) \wedge R = XV(m,l)) \vee \\ (a = ausleihe^{A_{sys}}(m,l) \wedge R = XA(m,l)) \vee \\ (a = rückgabe^{A_{sys}}(m) \wedge R = XR(m))\} \end{aligned}$$

Sei weiterhin F folgende Menge von Fairnessspezifikationen:

$$\begin{aligned} F =_{def} \{(a,R)_{fair}, a \in M_{sys}, R \in \wp([pcs_{fin}(EV, M_{sys})]) | \\ \exists m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}} . \\ (a = vormerkungswunsch^{A_{sys}}(m,l) \wedge R = FA(m,l)) \vee \\ (a = ausleihe^{A_{sys}}(m,l) \wedge R = FA(m,l)) \vee \\ (a = rückgabe^{A_{sys}}(m) \wedge R = FR(m))\} \end{aligned}$$

Q sei eine geeignete Menge von Kausalitäts- und Invariantenspezifikationen.

Dann definieren wir die Prozesssicht *BibPView* des Bibliothekssystems durch

$$BibPView =_{def} (BibGPView, F \cup Q)_P .$$

□

6.2 Spezifikation der Systemgrenze

Wie in Abschnitt 2.1 motiviert, gehen wir von einer umfassenden Sicht der fachlichen Aufgabenstellung, die mit Unterstützung des (zu entwickelnden) Softwaresystems zu lösen ist, aus. Umfassend bedeutet, dass sowohl solche Anteile der Aufgabenstellung, die durch das Softwaresystem zu realisieren sind, erfasst werden als auch Anteile, die wir der Umgebung des Softwaresystems zuordnen. Diese umfassende Betrachtung der Aufgabenstellung hilft zum einen für ein Verständnis der Rolle des Softwaresystems in seinem fachlichen Kontext, insbesondere für das Zusammenwirken von Softwaresystem und Umgebung zur Aufgabenlösung. Zum zweiten schaffen wir damit die Möglichkeit, im Systemmodell die (Festlegung einer) Aufgabenverteilung zwischen Software und Umgebung zu erfassen. Im all-

gemeinen ist diese Aufgabenverteilung nicht fest vorgegeben, sondern stellt vielmehr in vielen Fällen eine wichtige Entwurfsentscheidung und einen bedeutsamen Entwicklungsschritt dar.

In diesem Abschnitt führen wir die Produktarten ein, die dazu dienen, die Grenze des Softwaresystems aus fachlicher Sicht festzulegen, das heißt, die zu realisierenden fachlichen Aufgaben auf Softwaresystem und Umgebung zu verteilen. Im Fall von Informationssystemen bedeutet dies zum einen, die zu verwaltende fachliche Information aufzuteilen, in den durch das Softwaresystem zu verwaltenden Anteil und den Anteil der Umgebung. Analog dazu sind die jeweiligen Anteile fachlicher Abläufe zu bestimmen.

Gemäss unserem Systemmodell ist daher anzugeben, welche Entitäten und welche Prozessereignisse durch Softwaresystem oder Umgebung zu realisieren sind. Diese (System-)Eigenschaften erfassen wir anhand der

- Homomorphismen hs_i, $i \in [1,...,n_{snw}]$, zwischen den Zustandsalgebren (vergleiche Definition 2.27), sowie den
- Interaktions-Zuordnungsabbildungen hi_i, $i \in [1,...,n_{snw}]$ (vergleiche Definition 2.28).

In Abschnitt 6.2.1 führen wir die Produktart zur Aufteilung der Entitätsmenge ein, und in Abschnitt 6.2.2 diejenigen zur Aufteilung von Prozessereignissen.

Neben der Aufteilung von Entitäten und Prozessereignissen, ergeben sich Eigenschaften der Softwaresystemgrenze auch aus den Interaktionsformen zwischen Software und Umgebung, im Zusammenhang mit gemeinsam zu realisierenden Prozessereignissen. Hierzu führen wir in Abschnitt 6.2.3 die Produktart der Nutzungsspezifikation ein.

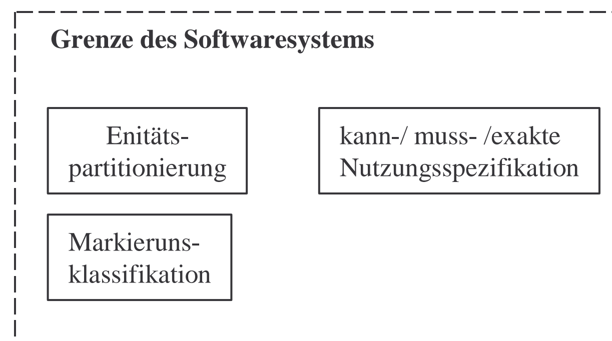


Abbildung 6.2: Produktarten zur Spezifikation der Grenze des Softwaresystems.

Abbildung 6.2 gibt einen Überblick über die Produktarten zur Spezifikation der Grenze des Softwaresystems.

6.2.1 Partitionierung der Entitäten

In unserem Systemmodell erfassen wir fachliche Information durch Entitäten (vergleiche Abschnitt 2.3.1). Daher beschreiben wir den Anteil an fachlicher Informa-

tion, der durch das zu entwickelnde Softwaresystem zu realisieren ist, durch eine Menge von Entitäten:

Definition 6.4 (Partitionierung der Entitäten)

Eine *Partitionierung der Entitäten* besteht aus

- einer endlichen Menge E´ von Entitätsidentifikatoren.

Die Semantik definieren wir durch

$$\left[\!\left[(E')_{E_dnw} \right]\!\right] =_{def} \{ sys \in SYS | \\ E' \subseteq E_{sys} \wedge \\ \forall i \in \left[1,...,(n_{snw})_{sys}\right]. \\ \left(\exists g \in states\left((dnw_i)_{sys} \right) \rightarrow (FS_{sys})_{State} \big|_{E'} . \\ \forall s \in \left((KS_i)_{sys} \right)_{State} . \\ \left(\left((hs_i)_{sys} \right)_{State} (s) \right) \Big|_{E'} = g\left(s|_{(dnw_i)_{sys}} \right) \right) \}$$

□

Eine Entitätspartitionierung $(E_d)_{E_dnw}$ legt mit E_d den Anteil der fachlichen Information/des Zustandsraums fest, der durch das Softwaresystem, im Systemmodell durch das Entwicklungsnetzwerk *dnw* (aus Definition 2.16) erfasst, zu realisieren ist. Das bedeutet, dass für jede spezifikatorische Sicht snw_i, $i \in [1,...,n_{snw}]$ auf das Realisierungsnetzwerk

$$nw = dnw \cup enw \text{ (vergleiche Definition 2.16)}$$

gelten muss, dass aus jedem Netzwerkzustand

$$s \in states(snw_i), i \in [1,...,n_{snw}]$$

die (realisierten) fachlichen Zustände der Entitäten aus E_d allein aus dem Software-Anteil, das heißt aus dnw_i, berechnet werden können müssen.

Zu beachten ist hierbei, dass wir im Realisierungsnetzwerk Zustände nur implizit erfassen, da nur das beobachtbare Interaktionsverhalten angegeben ist. Im Unterschied dazu, umfasst jede spezifikatorische Sicht snw_i einen expliziten Zustandsraum, den wir zu dem fachlichen Zustandsraum durch den jeweiligen Homomorphismus hs_i in Bezug setzen (vergleiche Abschnitt 2.6.1).

Weiterhin ist zu beachten, dass wir mit einer Partitionierung offen lassen, *wie* die Entitäten zu realisieren sind (Existenzquantifizierung von *g*), so dass keine Entwurfsentscheidungen vorweg genommen werden; das entspricht der *fachlichen* Sicht des Softwaresystems, da nur über Elemente der fachlichen Welt, die Entitäten, und das Entwicklungsnetzwerk als Einheit gesprochen wird, nicht über einzelne Komponenten des Softwaresystems und deren (technische) Zustandsräume. Diese fachliche Sicht entspricht der fachlichen (Anwender-)Perspektive, wodurch etwa die Produktart der *Anwenderanforderungen* des V-Modells [BDI97] charakterisiert ist, und wie wir sie in der Produktart der Anforderungsspezifikation in Ab-

schnitt 6.3 erfassen. Eine Entitätspartitionierung stellt ein Bindeglied zwischen der Spezifikation einer fachlichen Aufgabenstellung (vergleiche Abschnitt 6.1) und der Spezifikation der Zustandsrealisierung bezüglich einer bestimmten Softwarearchitektur (vergleiche Abschnitt 5.5.1) dar.

Zu beachten ist auch, dass wir als Voraussetzung annehmen, dass die Entitäten eine Granularität besitzen, die es erlaubt, sie entweder dem Entwicklungsnetzwerk oder der Umgebung vollständig zuzuordnen. Das bedeutet, dass Entitäten so feingranular sein müssen, dass sie nicht durch ein Zusammenwirken von Entwicklungsnetzwerk und Umgebung realisiert werden.

Mit Beispiel 6.3 geben wir die Entitätspartitionierung für unser Bibliothekssystem an:

Beispiel 6.3 (Entitätspartitionierung)

Die Anforderung, dass unser Bibliothekssoftwaresystem den gesamten fachlichen Zustandsraum zu realisieren hat, können wir mittels einer Entitätspartitionierung wie folgt formulieren:

$$\text{BibEPart} =_{\text{def}} (\text{p}_{\text{EPart}})_{\text{pred}}$$

mit

$$\text{p}_{\text{EPart}}(\text{sys}) \Leftrightarrow_{\text{def}} \text{sys} \in [\![(\text{E}_{\text{sys}})_{\text{E_dnw}}]\!].$$

für alle $\text{sys} \in \text{SYS}$.

Wie leicht zu sehen ist, erfüllt die Spezifikation aus Beispiel 5.5 diese Anforderung. In Beispiel 5.5 legen wir fest, dass die Komponente *BibDB* des Softwaresystems, den gesamten fachlichen Zustandsraum realisiert. Da

$$\text{BibDB} \in \text{dnw}_1$$

gilt, können wir eine Abbildung

$$\text{g} : \text{states}\big((\text{dnw}_1)_{\text{sys}}\big) \to (\text{FS}_{\text{sys}})_{\text{State}}$$

wobei für alle $\text{s} \in \text{states}\big((\text{dnw}_1)_{\text{sys}}\big)$ gelte

$$\text{g}(\text{s}) = \text{s}(\text{BibDB})$$

angeben.

□

6.2.2 Klassifikation der Ereignismarkierungen

In unserem Systemmodell ist die fachliche Sicht auf das Verhalten des betrachteten Anwendungssystems in Form von Prozessen gegeben (siehe Definition 2.12), wobei Prozesse aus Ereignissen und deren kausalen Abhängigkeiten aufgebaut sind. Demnach bedeutet die Festlegung der Rolle des zu entwickelnden Softwaresystems bezüglich der Realisierung fachlicher Abläufe, die Festlegung von Verantwortlich-

keiten des Softwaresystems für die Realisierung von Prozessereignissen. In Aktivitätsdiagrammen der UML geschieht dies beispielsweise in Form sogenannter *swimlanes* [OMG01].

Wir unterscheiden zwischen Prozessereignissen, die

- ausschließlich durch das Softwaresystem
- ausschließlich durch die Softwaresystem-Umgebung, oder
- durch Kooperation von Softwaresystem und Umgebung

realisiert werden. Diese Unterscheidungen können wir anhand der Interaktionszuordnungs-Abbildungen hi_i, $i \in [1,\ldots,n_{snw}]$, eines Entwicklungssystems formulieren, da wir durch diese Abbildungen die Realisierungsalternativen der verschiedenen Arten von Prozessereignissen, das heißt von Ereignismarkierungen, festlegen.

Formal definieren wir die drei genannten Markierungsklassen für jedes System $sys \in SYS$, wie folgt[29]:

- Menge der *Umgebungsmarkierungen*:

$$(M_{env})_{sys} =_{def} \{m \in M_{sys} \mid \forall i \in [1,\ldots,(n_{snw})_{sys}]. \exists p \in \left(IS_{ichannels((enw_i)_{sys}),states((enw_i)_{sys})}\right)^{\omega} \to \mathbb{B}. (hi_i)_{sys}(m) = \left\{\varphi \in ((IS_i)_{sys})^{\omega} \mid p\left(\varphi|_{ichannels((enw_i)_{sys}),states((enw_i)_{sys})}\right)\right\}\}$$

- Menge der *dnw-internen Markierungen*:

$$(M_{dnw})_{sys} =_{def} \{m \in M_{sys} \mid \forall i \in [1,\ldots,(n_{snw})_{sys}]. \exists p \in \left(IS_{ichannels((dnw_i)_{sys}),states((dnw_i)_{sys})}\right)^{\omega} \to \mathbb{B}. (hi_i)_{sys}(m) = \left\{\varphi \in ((IS_i)_{sys})^{\omega} \mid p\left(\varphi|_{ichannels((dnw_i)_{sys}),states((dnw_i)_{sys})}\right)\right\}\}$$

- Menge der *Schnittstellenmarkierungen*:

$$(M_{ss_dnw})_{sys} =_{def} \{m \in M_{sys} \mid \neg(m \in (M_{env})_{sys} \vee m \in (M_{dnw})_{sys})\}$$

wobei für jedes System $sys \in SYS$ und jedes $i \in [1,\ldots,(n_{snw})_{sys}]$

---

[29] bezüglich dnw_i, enw_i, sowie IS_i siehe Definition 2.19 und Definition 2.20.

$$\text{ichannels}\big((\text{dnw}_i)_{sys}\big) =_{def} \big\{c \in CH \big| \big(c \in \text{channels}(\text{dnw}_i)_{sys}\big) \wedge \neg\big(c \in \text{channels}(\text{enw}_i)_{sys}\big)\big\}$$

$$\text{ichannels}\big((\text{enw}_i)_{sys}\big) =_{def} \big\{c \in CH \big| \big(c \in \text{channels}(\text{enw}_i)_{sys}\big) \wedge \neg\big(c \in \text{channels}(\text{dnw}_i)_{sys}\big)\big\}$$

die Menge der internen Kanäle des Entwicklungs- beziehungsweise des Umgebungsanteils/-netzwerkes eines Spezifikationsnetzwerkes $(\text{snw}_i)_{sys}$ sei.

Zudem bezeichnen wir für jeden IS-Strom

$$\varphi \in \big((\text{IS}_i)_{sys}\big)^{\omega}$$

mit

$$\varphi|_{\text{ichannels}((\text{enw}_i)_{sys}),\text{states}((\text{enw}_i)_{sys})}$$

die Projektion auf die Kanalbelegungen der internen Kanälen des enw_i sowie den Zustandsanteil des Umgebungsanteils enw_i:

$$\forall \varphi \in \big((\text{IS}_i)_{sys}\big)^{\omega}.$$
$$\Big(\Pi_I\Big(\varphi|_{\text{ichannels}((\text{enw}_i)_{sys}),\text{states}((\text{enw}_i)_{sys})}\Big) = \Pi_I(\varphi)|_{\text{ichannels}((\text{enw}_i)_{sys})}\Big) \wedge$$
$$\Big(\Pi_S\Big(\varphi|_{\text{ichannels}((\text{enw}_i)_{sys}),\text{states}((\text{enw}_i)_{sys})}\Big) = \Pi_S(\varphi)|_{(\text{enw}_i)_{sys}}\Big)$$

Analoges gelte für $\varphi|_{\text{ichannels}((\text{dnw}_i)_{sys}),\text{states}((\text{dnw}_i)_{sys})}$.

Hinter obigen Definitionen steht die Vorstellung, dass es für die Angabe der Realisierungsalternativen von Ereignissen, die ausschließlich durch das Softwaresystem zu realisieren sind, ausreicht, Aussagen über Interaktionen auf Kanälen des Softwaresystems und über die Zustände der Komponenten des Softwaresystems zu machen. Das zu entwickelnde Softwaresystem modellieren wir im Systemmodell, wie in Definition 2.16 festgelegt, durch das Netzwerk *dnw*, die spezifikatorischen Sichten auf *dnw* durch die dnw_i, $i \in [1,...,n_{snw}]$ eines Entwicklungssystems.

Analog dazu beziehen wir uns bei Ereignissen, die ausschließlich durch die Umgebung realisiert werden, nur auf Interaktionen und Zustände von Komponenten der enw_i, $i \in [1,...,n_{snw}]$.

Die Menge der sogenannten Schnittstellen-Ereignisse ist dementsprechend die Menge der Ereignisse, die weder ausschließlich durch das Softwaresystem, noch ausschließlich durch dessen Umgebung realisiert werden.

Zur Klassifikation aller Ereignismarkierungen einer Entwicklung führen wir die folgende Art von Entwicklungsprodukten ein:

Definition 6.5 (Klassifikation von Ereignismarkierungen)

Eine *Klassifikation der Ereignismarkierungen* besteht aus

- drei endlichen Markierungsmengen M_d, M_e, M_{ss}.

Die Semantik definieren wir durch

$$\llbracket (M_d, M_e, M_{ss})_{M_dnw} \rrbracket =_{def} \{sys \in SYS \mid M_d \cup M_e \cup M_{ss} = M_{sys} \wedge M_d = (M_{dnw})_{sys} \wedge M_e = (M_{env})_{sys} \wedge M_{ss} = (M_{ss_dnw})_{sys}\}$$

Damit ist, wegen $M_{sys} = (M_{env})_{sys} \cup (M_{dnw})_{sys} \cup (M_{ss_dnw})_{sys}$, auch die Menge $(M_{ss_dnw})_{sys}$ eindeutig festgelegt.

□

Mit einer Markierungsklassifikation legen wir Verantwortlichkeiten und damit Anforderungen an das Verhalten eines Softwaresystems aus *rein fachlicher Sicht* fest, da wir mit Ereignismarkierungen Bezug auf Elemente einer fachlichen Aufgabenstellung nehmen, und die Markierungen unabhängig von einer konkreten Netzwerkstruktur (der Realisierung) von Softwaresystem und Umgebung klassifizieren. Daher stellt die Markierungsklassifikation ein geeignetes Bindeglied zwischen der Spezifikation einer fachlichen Aufgabenstellung und der Spezifikation der Realisierungsalternativen von Ereignissen bezüglich einer konkreten Software-Architektur, und damit dem Softwaresystem-Entwurf, dar. Damit ist eine Markierungsklassifikation ein passender Bestandteil einer Anforderungsspezifikation (vergleiche Abschnitt 6.3).

Im folgenden Beispiel legen wir Verantwortlichkeiten für die Realisierung von Ereignissen fest, die im Rahmen einer Medien-Vormerkung unseres Bibliotheksbeispiels auftreten.

Beispiel 6.4 (Klassifikation einer Ereignismarkierung)

Wir beziehen uns auf die Bearbeitung von Vormerkungswünschen für Medien, wie wir sie in Beispiel 4.1 behandelt haben. Sei ACT die algebraische Spezifikation aus Beispiel 4.1, durch die wir die Ereignismarkierungen einer Vormerkungsbearbeitung einführen. Wir betrachten Ereignismarkierungen der Form

m.vormerken*(l),

die für die entsprechende Zustandsänderung der Medien-Entität stehen. Wir gehen davon aus, dass die Verwaltung der Medien einer Bibliothek Aufgabe des zu entwickelnden Bibliotheks-Softwaresystems ist. Daher zeichnen wir mit folgendem Entwicklungsprodukt ausschließlich das Softwaresystem für die Realisierung von

m.vormerken*(l)

verantwortlich:

$(p_{VMClass})_{pred}$

mit

$$p_{VMClass}(sys) \Leftrightarrow_{def} sys \in [\![(\{Medium, Leser, Status\}\Sigma_{ACT}, C_{ACT})_{muss_red}]\!] \wedge$$
$$\forall m \in (A_{sys})_{id_{Medium}}, (A_{sys})_{id_{Leser}} .$$
$$m.vormerken * (1) \in (M_{dnw})_{sys}$$

für alle $sys \in SYS$, wobei wir, wie oben definiert, mit $(M_{dnw})_{sys}$ die Menge der dnw-internen Ereignismarkierungen eines Systems $sys \in SYS$ bezeichnen. Die Klassifikation als dnw-internes Ereignis deutet darauf hin, dass die Eintragung einer Vormerkung nicht direkt an der Schnittstellen des Softwaresystems angestoßen werden kann.

Eine weitere Form von Markierung, die in einer Vormerkungsbearbeitung auftreten kann, ist

rückgabeaufforderung(m)

für einen Medienidentifikator *m*. Wie in Beispiel 4.1 beschrieben, steht ein mit

rückgabeaufforderung(m)

markiertes Ereignis unter anderem dafür, den aktuellen Entleiher des Mediums *m* per E-Mail zur Rückgabe aufzufordern. Diese Ereignisform sehen wir als ein Beispiel dafür, dass die Festlegung von Verantwortlichkeiten im allgemeinen nicht durch fachliche Notwendigkeiten vorbestimmt ist, sondern Raum für Entwurfsentscheidungen lässt:

- Wird ein automatischer E-Mail-Versand gewünscht und soll die dafür notwendige Software entwickelt werden, so ist ausschließlich das Bibliotheks-Softwaresystem für die Realisierung einer Rückgabeaufforderung verantwortlich.
- Soll das Softwaresystem eine Bibliotheksangestellte auf die Notwendigkeit einer Rückgabeaufforderung hinweisen, beispielsweise durch eine entsprechende Meldung in einem Bildschirm-Dialog, und sei die E-Mail dann durch die Angestellte mit Hilfe eines (existierenden) E-Mail-Programms zu versenden, so handelt es sich um ein Schnittstellen-Ereignis.
- Muss eine Bibliotheksangestellte die Notwendigkeit einer Rückgabeaufforderung ohne Unterstützung des Softwaresystems erkennen, so verstehen wir *rückgabeaufforderung(m)* als ein Umgebungsereignis.

□

Analog zu der oben, bezüglich dem Entwicklungsnetzwerk eines Systems formulierten Klassifikation, lassen sich Markierungen noch feiner bezüglich einzelner Komponenten klassifizieren. Auf diese Weise kann die Festlegung von Rollenverantwortlichkeiten der Geschäftsprozessmodellierung in unserem Ansatz abgebildet werden. In erweiterten EPKs (Ereignisgesteuerten Prozessketten) (vergleiche [Schee98]) werden solche Verantwortlichkeiten durch Kanten zwischen Organisationseinheiten und sogenannten Funktionen, die im Wesentlichen markierten Ereignissen unseres Prozessbegriffs entsprechen, dargestellt. Die Aktivitätsdiagramme der UML stellen zu diesem Zweck die sogenannten *swimlanes* zur Verfügung [OMG01].

6.2.3 Nutzungsspezifikation des Softwaresystems

Ein Softwaresystem übernimmt fachliche Aufgaben, indem es Ereignisse fachlicher Abläufe realisiert, entweder mit oder ohne Beteiligung der Softwaresystem-Umgebung. In Abschnitt 6.2.2 führen wir die Produktarten zur Festlegung von Verantwortlichkeiten für die Realisierung von Prozessereignissen ein. Diese Art von Entwicklungsprodukten lässt jedoch offen, *wie* Softwaresystem und Umgebung interagieren, um Ereignisse zu realisieren. Dies zu spezifizieren bedeutet, die (Interaktions-)Formen/Protokolle zwischen dem Softwaresystem-Nutzer und dem Softwaresystem zu charakterisieren, was Gegenstand der im folgenden eingeführten Produktarten, den sogenannten Nutzungsspezifikationen, ist.

Nutzungsspezifikationen sind bedeutsam, da sie einerseits Anleitung dafür sind, wie das Softwaresystem zu nutzen ist, und andererseits Anforderungen an die Schnittstellen-Eigenschaften des Softwaresystems darstellen. Beispielsweise lassen sich hier Anforderungen an (graphische) Benutzeroberflächen einordnen (siehe auch Beispiel 6.5).

Unserem Systemmodell liegt die Vorstellung zugrunde, dass das *beobachtbare* Verhalten einer Komponente, beziehungsweise eines Komponenten-Netzwerkes, auf Interaktionen an der (syntaktischen) Schnittstelle beschränkt ist. Das heißt, auf Komponenten- beziehungsweise Netzwerkzustände kann nur implizit durch Interaktion zugegriffen werden, und Interaktionen auf netzwerkinternen Kanälen sind nicht Teil des beobachtbaren Verhaltens.

Aus Sicht des Nutzers eines Softwaresystems ist das beobachtbare Verhalten des Softwaresystems relevant. Daher beschränken wir uns in Nutzungsspezifikationen auf diesen Aspekt. Durch diese Beschränkung auf die Außensicht eines Softwaresystems (Black-Box-Sicht) bleiben Nutzungsspezifikationen unabhängig von internen Softwaresystemaspekten, wodurch etwa die Vermischung von Nutzungsspezifikation und Architekturspezifikation verhindert wird.

Jede Nutzung eines Softwaresystems dient einem bestimmten fachlichen Zweck, den wir in Form des jeweils realisierten Prozessereignisses erfassen. Beispielsweise dient die Interaktion zur Realisierung eines mit

vormerkungswunsch(m,l)

markierten Ereignisses dazu, einen Vormerkungswunsch zu äußern und damit die Bearbeitung einer Vormerkung für ein Medium *m* und einen Leser *l* auszulösen (vergleiche Geschäftsprozess-Spezifikation aus Beispiel 4.1).

Aufgrund des Zusammenhangs zwischen fachlichem Zweck, modelliert durch eine Ereignismarkierung, und den Nutzungsformen des Softwaresystems zur Erreichung dieses Zwecks, betrachten wir Nutzungsspezifikationen als Anforderungen an die Realisierung von Prozessereignissen. Im Systemmodell erfassen wir die Realisierung von Prozessereignissen durch die Interaktions-Zuordnungs-Abbildungen hi_i, für jedes Spezifikationsnetzwerk snw_i, $i \in [1,...,n_{snw}]$.

Da wir eine Nutzungsspezifikation als Anforderung an das zu entwickelnde Softwaresystem verstehen, im Systemmodell modelliert durch das Netzwerk *dnw* (siehe Definition 2.16), müssen die geforderten Eigenschaften für alle spezifikatorischen Sichten auf *dnw*, das heißt *für alle* dnw_i, $i \in [1,...,n_{snw}]$, gelten.

Die drei genannten Charakteristika von Nutzungsspezifikationen,

- Beschränkung auf das beobachtbare Verhalten des (zu entwickelnden) Softwaresystems,
- Bezug der spezifizierten Nutzungs-/Interaktionsformen zu einem fachlichen Zweck, gegeben in Form einer Ereignismarkierung, sowie
- Anforderung an das Softwaresystem und nicht (nur) an eine bestimmte spezifikatorische Sicht,

bestimmen die Form der folgenden Definition der Produktart der Nutzungsspezifikationen:

Definition 6.6 (Nutzungsspezifikationen des Softwaresystems)

Eine *Nutzungsspezifikation* des Softwaresystems besteht aus

- einer Ereignismarkierung *m* sowie
- einer Menge $X \subseteq \left(\vec{C^*}\right)^\omega, C \subseteq CH$.

Die Semantik einer *exakten* (hi_dnw) *Nutzungsspezifikation* ist wie folgt:

$$\llbracket (m,X)_{hi_dnw} \rrbracket =_{def} \{sys \in SYS | \\ C \subseteq (in(dnw_{sys}) \cup out(dnw_{sys})) \wedge \\ \forall i \in [1,...,(n_{snw})_{sys}].X = IStreams((hi_i)_{sys}(m))|_C \}$$

Analog dazu gilt für eine *kann*-Nutzungsspezifikation $(hi_dnw \supseteq)$

$$IStreams((hi_i)_{sys}(m))|_C \supseteq X$$

und für eine *muss-Spezifikation* $(hi_dnw \subseteq)$

$$IStreams((hi_i)_{sys}(m))|_C \subseteq X$$

wobei für alle $m \in M_{sys}, i \in [1,...,(n_{snw})_{sys}]$ gelte:

$$IStreams(hi_{i_{sys}}(m))|_C =_{def} \left\{ \psi \in \left(\vec{C^*}\right)^\omega \middle| \exists \varphi \in hi_{i_{sys}}(m). \\ \psi = (\Pi_{Istream}(\varphi))|_C \right\}$$

Kann-Nutzungsspezifikationen bezeichnen wir synonym auch als *Nutzungsszenarien*.

□

Die Beschränkung von Nutzungsspezifikationen auf das *beobachtbare* Softwaresystemverhalten spiegelt sich in obiger Definitionen dadurch wider, dass wir mit einer Nutzungsspezifikation nur Eigenschaften

- der Interaktionshistorien von Kanälen, die Teil der syntaktischen Schnittstelle des Softwaresystems sind ($C \subseteq \left(in\left(dnw_{sys}\right) \cup out\left(dnw_{sys}\right)\right)$ und $X = IStreams(...)|_C$), und
- keine Eigenschaften der Zustandshistorien, $IStreams(...) =_{def} \left\{\psi \in \left(\vec{C^*}\right)^{\omega} \middle| ...\psi = \left(\Pi_{Istream}(...)\right)\right\}$,

von Ausführungsfolgen der Spezifikationsnetzwerke festlegen.

Das folgende Beispiel gibt ein Nutzungsszenario und eine exakte Nutzungsspezifikation für unser Bibliothekssystem an, und skizziert den Zusammenhang der Nutzungsspezifikation mit der Spezifikation graphischer Benutzeroberflächen:

Beispiel 6.5 (Nutzungsspezifikation der Bibliothek)

In Beispiel 4.1 führen wir

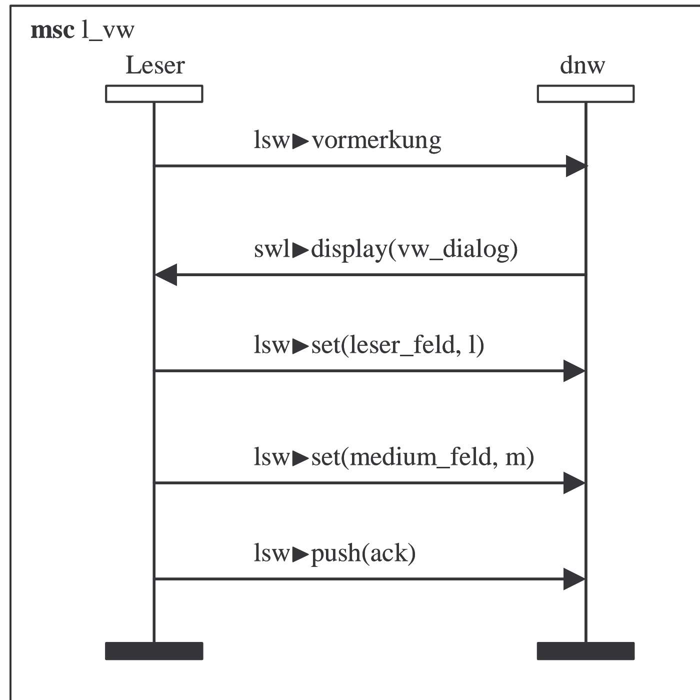
$$vormerkungswunsch : id_{Medium} \times id_{Leser} \rightarrow Act$$

als Konstruktor von Ereignismarkierungen ein, welche die Äußerung eines Vormerkungswunsches für ein Medium m und einen Leser l repräsentieren (*Leser* und *Medium* seien Wertsorten, wie etwa in Beispiel 3.1 formuliert). Wir gehen in diesem Beispiel, gemäss der informellen Aufgabenstellung aus Abschnitt 1.7, davon aus, dass Vormerkungswünsche sowohl von Lesern als auch von Bibliotheksangestellten geäußert werden können. In Beispiel 5.1 modellieren wir die (Nutzer-)Rolle des Lesers durch die Komponente *Leser* und analog dazu die Rolle der Bibliotheksangestellten durch die Komponente *BibAng*. Beide Komponenten sind mit dem Softwaresystem, repräsentiert durch das Netzwerk *dnw*, durch jeweils ein Kanalpaar, *lsw*, *swl* beziehungsweise *bsw*, *swb*, verbunden. Dementsprechend wird die Äußerung eines Vormerkungswunsches durch Interaktion auf einem der beiden Kanalpaare realisiert. Wir legen folgendes Interaktionsprotokoll fest:

- In einem ersten Schritt zeigt der Nutzer an, dass er einen Vormerkungswunsch äußern will, zum Beispiel in einem Auswahl-Menü, dass alle, durch das Softwaresystem angebotenen Dienste, das heißt alle unterstützten Arten von Prozessereignissen, anzeigt.
- Daraufhin zeigt das Softwaresystem einen Dialog an, welcher die Eingabe eines Medien- und eines Leser-Identifikators ermöglicht sowie deren Bestätigung durch aktivieren eines Schalters.
- Nach der Eingabe beider Werte (in beliebiger zeitlicher Reihenfolge) bestätigt der Nutzer diese durch aktivieren des Schalters, womit die Äußerung eines Vormerkungswunsches erfolgreich abgeschlossen sei.

Das in Abbildung 6.3 dargestellte Sequenzdiagramme zeigt die Nutzungsalternative, in der ein Leser einen Vormerkungswunsch äußert, und dabei zuerst den Identifikator des vorzumerkenden Lesers eingibt und anschließend den des Mediums.

Somit kann das Sequenzdiagramm als Darstellung eines Nutzungsszenarios verstanden werden. Der durch das Softwaresystem anzuzeigende Dialog ist sowohl als Parameter *vw_dialog* als auch graphisch dargestellt. Er setzt sich aus zwei (beschrifteten) Eingabefeldern und einem Bestätigungs-Schalter zusammen.



vw_dialog =
< (input, medium_feld, idMedium,"Medien-ID"),
(input, leser_feld, idLeser,"Leser-ID"),
(schalter, ack, "ok") >

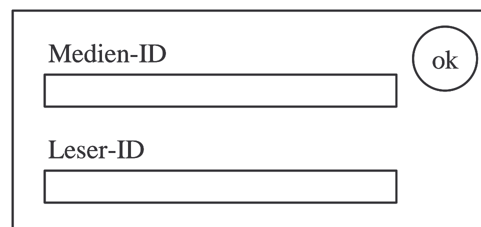


Abbildung 6.3: Eine Nutzungsalternative zur Äußerung eines Vormerkungswunsches.

Die Menge der exakten Nutzungsspezifikationen von Vormerkungswunsch-Äusserungen geben wir wie folgt an:

$$\forall m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}} .$$

$$\left(\text{vormerkungswunsch}^{A_{sys}}(m,l), XV(m,l)\right)_{hi_dnw}$$

mit

$C =_{def} \{lsw, swl, bsw, swb\}$

und

$$XV(m,l) =_{def} \Big\{\varphi \in \big(\vec{C^*}\big)^{\omega} \Big| \exists j_1, j_2, ..., j_5 \in \mathbb{N}_+, ci, co \in C. \\ ((ci = lsw \wedge co = swl) \\ \vee (ci = bsw \wedge co = swb)) \wedge \\ (j_1 < j_2 < j_3) \wedge (j_2 < j_4) \wedge (j_3 < j_5 \wedge j_4 < j_5) \wedge \\ in(vormerkung, (\varphi @ j_1)(ci)) \wedge \\ in(display(vw_dialog, (\varphi @ j_2)(co)) \wedge \\ in(set(leser_feld, l), (\varphi @ j_3)(ci)) \wedge \\ in(set(medium_feld, m), (\varphi @ j_4)(ci)) \wedge \\ in(push(ack), (\varphi @ j_5)(ci))\Big\}$$

wobei $sys \in SYS$.

□

Zu beachten ist, dass wir einen möglichen Bezug zwischen (dem Entwurf graphischer) Benutzer-Oberflächen und unserem Kanal- und interaktionsbasierten Netzwerk- und Komponentenmodell herstellen, indem wir graphische Dialogfenster und deren Anzeigen (abstrakt) durch entsprechende Nachrichtenformen und deren Ausgabe modellieren, in obigem Beispiel etwa durch *co►display(vw_dialog)*. Analog dazu modellieren wir auch Benutzereingaben durch entsprechende Nachrichten und Interaktionen.

Obiges Beispiel macht auch deutlich, dass die Verknüpfung von Geschäftsprozess- und Nutzungsspezifikation via Ereignismarkierungen, zu einer modularen, an der fachlichen Verhaltenssicht orientierten Struktur von Nutzungsspezifikationen und gegebenenfalls von Benutzeroberflächen führt. So ergeben sich beispielsweise die wesentlichen Aspekte der Nutzung des Bibliotheks-Softwaresystems im Zusammenhang mit einer Vormerkung, aus den Nutzungsspezifikationen für die einzelnen Ereignisarten, die in der Spezifikation des Vormerkungs-Geschäftsprozesses vorkommen.

Eine (exakte) Nutzungsspezifikation ist ähnlich zu dem (informellen) Begriff des *Use Case* aus [Jac92, JBR99]. In [Jac92, JBR99] wird ein *Use Case* unter anderem dadurch charakterisiert, dass er die möglichen Interaktionsformen zwischen dem Softwaresystem und einem Aktor beschreibt, die für den Aktor einen bestimmten Zweck erfüllen/Nutzen bringen:

> *“A use case specifies a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor.”* [JBR99]

In unserem Ansatz erfassen wir diesen Zweck/Nutzen mittels der Ereignismarkierung, auf die sich eine Nutzungsspezifikation bezieht. Typischerweise wird durch

Use Cases, analog zu unseren Nutzungsspezifikationen, nur das Interaktionsverhalten an der Schnittstelle des betrachteten Softwaresystems beschrieben. Mögliche systeminterne Interaktionen sind nicht Bestandteil eines *Use Case*, sondern vielmehr der (Beschreibung der) *Use Case Realization* (siehe [JBR99]).

In Abschnitt 6.3 definieren wir Nutzungsspezifikationen als Baustein der Produktart der Anforderungsspezifikation.

6.3 Anforderungsspezifikation

Die Anforderungsspezifikation, auch als *Produkt-Definition* [Bal96], *Pflichtenheft* [Bal96], *Software Requirements Specification* [ANS84, DT90] und *Anwenderforderungen* [BDI97] bezeichnet, ist eine Produktart, die in nahezu allen Vorgehensmodellen und Entwicklungsstandards zu finden ist. Wir definieren im Folgenden eine Form der Anforderungsspezifikation, die auf der (System-)Modellvorstellung basiert, die wir in Kapitel 2 einführen. Hierzu komponieren wir eine Anforderungsspezifikation aus den in Abschnitt 6.1 bis Abschnitt 6.2 eingeführten Produktarten.

Allgemein wird von einer Anforderungsspezifikation gefordert, dass sie Anforderungen an das zu entwickelnde Softwaresystem aus fachlicher und aus Nutzersicht vollständig angibt. Dies bedeutet insbesondere, dass Eigenschaften des Softwaresystems in *fachlichen* Begriffen/Modellierungskonzepten, das heißt in Begriffen der Anwendungswelt, anzugeben sind. Zudem werden (aus software-technischer Sicht) nur Eigenschaften der Außensicht (Black-Box-Sicht) des Softwaresystems angegeben, das heißt nur strukturelle Eigenschaften der (syntaktischen) Schnittstelle des Softwaresystems und Eigenschaften des Verhaltens an dieser Schnittstelle. Abstrakt wird dies häufig mit der Forderung ausgedrückt, dass eine Anforderungsspezifikation das WAS ohne das WIE anzugeben hat (vergleiche beispielsweise die Produktart der Anwenderforderungen des V-Modells [BDI97] und die Produkt-Definition in [Bal96]).

In unserem Ansatz bedeutet obige Forderung zum einen, wie in Abschnitt 6.2.1 und Abschnitt 6.2.2 diskutiert, anhand einer fachlichen Aufgabenstellung Verantwortlichkeiten des Softwaresystems festzulegen (fachliche Sicht des Softwaresystems). Zum zweiten umfasst eine Anforderungsspezifikation die Nutzersicht mittels den, in Abschnitt 6.2.3 eingeführten, Nutzungsspezifikationen für alle die fachlichen Aufgaben, die durch Nutzung des Softwaresystems, das heißt durch Kooperation von Umgebung und Softwaresystem, zu realisieren sind.

Die Forderung nach Vollständigkeit der Inhalte einer Anforderungsspezifikation können wir erfüllen, wenn wir sowohl die fachliche als auch Nutzersicht durch die Produktarten erfassen, die das Kriterium der Vollständigkeit erfüllen.

Abbildung 6.4 gibt einen Überblick der Produktarten, die wir als Elemente einer Anforderungsspezifikation wählen.

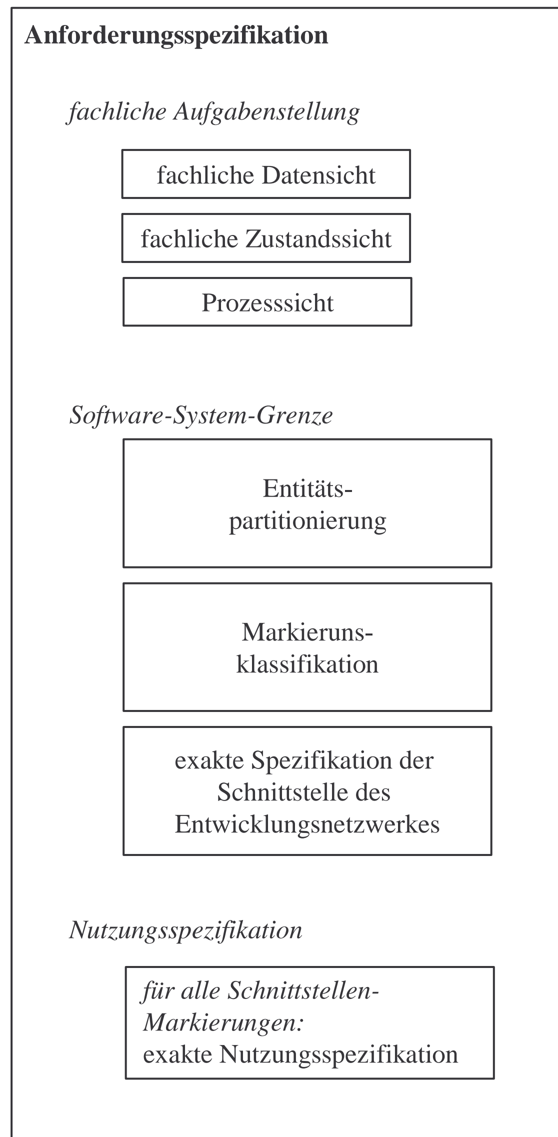


Abbildung 6.4: Struktur einer Anforderungsspezifikation.

Die Zusammenfassung der genannten Produktarten zur Produktart der Anforderungsspezifikation ist aufgrund der inhaltlichen Kohärenz der genannten Bestandteile sinnvoll:

> Um eine Softwaresystem-Grenze (Aufteilung fachlicher Verantwortlichkeiten auf Softwaresystem und Umgebung) spezifizieren zu können, muss eine (umfassende) fachliche Aufgabenstellung gegeben sein. Eine Nutzungsspezifikation stellt wiederum eine Konkretisierung hinsichtlich der Kooperation von Softwaresystem und Umgebung zur Realisierung gemeinsamer fachlicher Aufgaben dar.

Aufgrund der Beschränkung auf die fachliche und die Nutzersicht einerseits, und deren vollständige Erfassung andererseits, ist eine Anforderungsspezifikation für folgende, für eine Softwareentwicklung bedeutsame Zwecke geeignet:

- Aufgrund der Vollständigkeit stellt eine Anforderungsspezifikation ein Referenzprodukt
  - für Fragen der fachlichen und der Nutzersicht, sowie
  - für die Abnahme/Validierung und das Testen/Verifikation

  eines Softwaresystems dar.
- Im Forward-Engineering ist sie eine geeignete Grundlage für den Software-Entwurf, ebenso wie
- für die Entwicklung und Auswahl von Software(architektur)-Alternativen: da eine Anforderungsspezifikation das WAS ohne das WIE angibt, können einem WAS mehrere alternative WIE, das heißt Softwarearchitekturen, gegenüber gestellt werden.
- Durch die Erfassung der fachlichen Rolle des Softwaresystems enthält eine Anforderungsspezifikation wichtige Informationen für die Weiterentwicklung/Anpassung eines Softwaresystems bezüglich einer veränderten fachlichen Aufgabenstellung. Eine Anforderungsspezifikation des existierenden und zu verändernden Systems kann in diesem Zusammenhang als Ergebnis einer IST-Analyse verstanden werden.
- Eine Anforderungsspezifikation gibt Anleitung für die Nutzung des Softwaresystems, das heißt auf welche Weise Dienste/Funktionalität des Softwaresystems genutzt werden können.
- Sie ist geeignet als Vertragsbestandteil zwischen Auftraggeber und Auftragnehmer, denn im allgemeinen ist ein Auftraggeber an der Erreichung fachlicher Ziele, durch den Einsatz des Softwaresystems, interessiert, so dass er eine fachliche Sicht einnimmt.

Allgemein gilt, dass durch die Beschreibung der fachlichen und der Nutzersicht ohne Bezugnahme auf software-technische Aspekte, wie sie gegeben ist, wenn im Rahmen einer Entwicklung eine Anforderungsspezifikation (in obigem Sinne) erstellt wird, klar unterschieden werden kann, zwischen fachlichen und software-technischen Notwendigkeiten. Zudem wird dadurch Überspezifikation vermieden. Beides trägt zu einem klaren Verständnis der fachlichen Rolle eines Softwaresystems bei.

Definition 6.7 (Anforderungsspezifikation)

Eine Anforderungsspezifikation besteht aus

- einer Spezifikation der fachlichen Aufgabenstellung, wiederum bestehend aus
  - § einer exakten Spezifikation der Entitätsrechenstrukturen (synonym Datensicht) $(W, \Sigma, C)_{fe}$ (gemäss Definition 3.5)
  - § einer Zustandssicht $\{(X)_E, (s_0)_{init_state}\}$ (gemäss Definition 6.1)
  - § einer Prozesssicht $(G, Q)_P$ (gemäss Definition 6.3)

- einer Spezifikation der Softwaresystem-Grenze, bestehend aus
  - § einer Entitätspartitionierung $(X')_{E_dnw}$ (gemäss Definition 6.4), mit $X' \subseteq X$
  - § einer Markierungsklassifikation $(M_i, M_e, M_{SS})_{M_dnw}$ (gemäss Definition 6.5)
  - § einer exakten Schnittstellenspezifikation des Softwaresystems $(I,O)_{ss_dnw}$ (gemäss Definition 5.7)
- einer Spezifikation der Nutzungssicht, bestehend aus
  - § einer M_{ss}-indizierten Menge exakter Nutzungsspezifikationen $\left\{(m, X_m)_{nf_dnw} \middle| m \in M_{ss}\right\}$ (gemäss Definition 6.6)

Die Semantik einer Anforderungsspezifikation ergibt sich aus der Semantik der Produktmenge, für die sie steht.

□

Zu beachten ist, dass wir, aufgrund des Anspruchs auf Vollständigkeit einer Anforderungsspezifikation, in der Nutzungssicht für *jede* in der Spezifikation der Systemgrenze festgelegten Schnittstellenmarkierung, eine *exakte* Nutzungsspezifikation fordern.

Im folgenden Beispiel geben wir die Anforderungsspezifikation für unser Bibliothekssystem an.

Beispiel 6.6 (Anforderungsspezifikation des Bibliothekssystems)

Mit Beispiel 6.1 bis Beispiel 6.5 sind bereits einige Bausteine einer Anforderungsspezifikation für unser Bibliothekssystem gegeben:

BibAnfSpez $=_{def}$ {BibDataView, BibStateView, BibPView,
BibEPart, BibMClass, BibSWSS,
BibUseSpec}

Dabei seien *BibDataView* und *BibStateView* die Daten- beziehungsweise die Zustandssicht aus Beispiel 6.1, *BibPView* die Prozesssicht aus Beispiel 6.2, und die Entitätspartitionierung *BibEPart* aus Beispiel 6.3.

Die exakte Spezifikation der Schnittstelle des Bibliotheks-Softwaresystems, erfasst im Systemmodell durch das Entwicklungsnetzwerk *dnw*, erweitert die in Beispiel 5.1 gegebene Schnittstelle um ein Kanalpaar, welches das Softwaresystem mit einer Komponente *E-Mailer* verbindet, die den Versand von E-Mails realisiere (vergleiche Abbildung 6.5):

$$\text{BibSWSS} =_{def} (\{lsw, bsw, esw\}, \{swl, swb, swe\})_{SS_dnw}$$

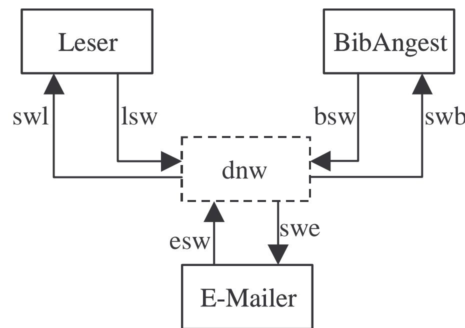


Abbildung 6.5: exakte Schnittstelle des Bibliotheks-Softwaresystems

Weiterhin sei *BibMClass* eine geeignete Markierungsklassifikation (vergleiche Beispiel 6.4) und *BibUseSpec* eine entsprechende Menge exakter Nutzungsspezifikationen aller Schnittstellen-Markierungen, etwa in Form der Nutzungsspezifikation für Vormerkungswünsche aus Beispiel 6.5.

□

6.3.1 Verwandte Produktarten

Die mit Definition 6.7 gegebene Produktart der Anforderungsspezifikation deckt den Kern eines Pflichtenheftes nach [Bal96] ab, was wiederum große Ähnlichkeit zu der in [ANS84] definierten *Software Requirements Specification* hat. Ebenso werden wesentliche Elemente der Produktart der Anwenderforderungen des V-Modells [BDI97] erfasst. Damit stellt unsere Anforderungsspezifikation eine präzise Konkretisierung von Kerninhalten gängiger Standards im Hinblick auf die Entwicklung verteilter Informationssysteme dar.

Ein Pflichtenheft wird in [Bal96] wie folgt charakterisiert:

> *„Aufgabe: Das Pflichtenheft enthält eine Zusammenfassung aller fachlichen Anforderungen, die das zu entwickelnde Software-Produkt aus Sicht des Auftraggebers erfüllen muss.*
>
> *[...]*
>
> *Inhalt: Fachlicher Funktions-, Daten-, Leistungs- und Qualitätsumfang des Produktes. Beschreibung des WAS, nicht des WIE.“*

Es gliedert sich grob folgendermaßen (wobei unter *Produkt* das zu entwickelnde Softwaresystem verstanden wird):

1. Zielbestimmung.
   Beschreibung welche Ziele durch den Einsatz des Produktes erreicht werden sollen; Festlegung von Muss-, Wunsch- und Abgrenzungskriterien.
2. Produkt-Einsatz.
   Angabe der Anwendungsbereiche, zum Beispiel Textverarbeitung im Büro, und der Zielgruppen, zum Beispiel Sekretärinnen und Schreibkräfte, des Produktes.

3. Produkt-Umgebung.
   Beschreibung der Umgebung des Produktes, unter anderem, welche Softwaresysteme und Hardware zur Verfügung stehen.
4. Produkt-Funktionen.
   Funktionale Beschreibung des Produktes aus Benutzersicht.
5. Produkt-Daten.
   Beschreibung der langfristig zu speichernden Daten aus Benutzersicht.
6. Produkt-Leistungen.
   Zeit- und umfangsbezogene Anforderungen, zum Beispiel maximaler Datendurchsatz und maximale Dialogantwortzeiten.
7. Benutzungsoberfläche.
   Anforderungen an die Benutzungsoberfläche, wie Bildschirmlayout.
8. Qualitäts-Zielbestimmung.
   Geforderte Qualitätsmerkmale des Produktes.
9. GlobaleTestszenarien.
   Anwendungsbezogene Testfälle, die im allgemeinen mehrere Produktfunktionen in Anspruch nehmen.
10. Entwicklungsumgebung.
    Beschreibung welche Soft- und Hardware, etc. für die Entwicklung des Produktes zur Verfügung stehen.
11. Ergänzungen.
    Spezielle Anforderungen, die nicht durch die vorangehenden Punkte abgedeckt sind, zum Beispiel Installationsbedingungen.

Mit einer Anforderungsspezifikation gemäss Definition 6.7 erfassen wir die Punkte *Zielbestimmung* und *Produkt-Einsatz* in gewisser Weise dadurch, dass wir mit der Spezifikation der fachlichen Aufgabenstellung eine ganzheitliche, fachliche Sicht einnehmen, die nicht auf das Softwaresystem beschränkt ist. Die *Produkt-Umgebung* können wir in unserem Ansatz erfassen, indem wir eine Umgebungsarchitektur (vergleiche Abschnitt 6.4) angeben. *Produkt-Funktionen* entsprechen den internen und den Schnittstellen-Ereignismarkierungen, wie sie durch eine Markierungsklassifikation ausgezeichnet werden. Nutzungsspezifikationen konkretisieren diese aus Anwendersicht. *Produkt-Daten* erfassen wir mittels der Entitätspartitionierung in Form der Entitätsmenge, die durch das Softwaresystem zu realisieren ist. Anhand von Beispiel 6.5 zeigen wir, wie durch Nutzungsspezifikationen sowohl Aspekte der *Produkt-Funktionen* aus Nutzersicht als auch der *Benutzungsoberfläche* integriert werden. Die Nutzungsspezifikationen unserer Form von Anforderungsspezifikation stellen globale Testszenarien dar, die gegebenenfalls um Testszenarien in Form von Netzwerk-Verhaltensspezifikationen (vergleiche Abschnitt 5.2), die sich auf das Umgebungsnetzwerk und die Außensicht des Softwaresystems/Entwicklungsnetzwerkes beschränken, ähnlich zur Beschränkung von Nutzungsspezifikationen auf die Außensicht des Softwaresystems, ergänzt werden können. Die Punkte *Produkt-Leistungen*, *Qualitäts-Zielbestimmung*, *Entwicklungsumgebung* und *Ergänzungen* stellen Erweiterungen des Pflichtenheftes gegenüber unserer Form der Anforderungsspezifikation dar.

In der Produktart der Anwenderforderungen des V-Modells wird gefordert

> *„Zur Festlegung der geforderten Systemfunktionalität werden Geschäftsprozesse definiert. Diese sind ganzheitlich, d. h. vollständig zu betrachten, auch wenn das geforderte System nur Ausschnitte davon unterstützt.“* [BDI97]

In unserem Ansatz erfüllen wir diesen Anspruch durch die umfassende und ganzheitliche Sicht, aus der wir die fachliche Aufgabenstellung spezifizieren, zusammen mit der Festlegung der Softwaresystem-Grenze *bezüglich dieser* ganzheitlichen, fachlichen Aufgabenstellung.

6.4 Architekturen

Im Laufe der (schrittweisen) Entwicklung eines Softwaresystems erfassen wir dessen Eigenschaften im allgemeinen in mehreren Sichten, beispielsweise auf verschiedenen Detaillierungsebenen oder anhand der horizontalen oder vertikalen Schichtung. Diese Sichten einer Entwicklung bilden wir in unserem Systemmodell durch die Menge der sogenannten Spezifikationsnetzwerke snw_i, mit ihrem jeweiligen Entwicklungs- und Umgebungsanteil dnw_i beziehungsweise enw_i (siehe Abschnitt 2.5.4) ab.

Unter dem im Folgenden eingeführten Begriff der *Architektur* verstehen wir eine Menge von Entwicklungsprodukten, durch welche eine dieser spezifikatorischen (Netzwerk-)Sichten snw_i, dnw_i oder enw_i, *vollständig* charakterisiert wird. Vollständigkeit bedeutet hierbei, dass die durch die spezifikatorische Sicht gestellten Anforderungen an das Realisierungsnetzwerk nw, eindeutig festgelegt sind. Damit eignet sich eine Architektur als Referenz für die jeweilige spezifikatorische Sicht, da garantiert ist, dass weitere Entwicklungsprodukte, die zu der gegebenen Architektur konsistent sind, keine zusätzlichen, nicht redundanten Anforderungen formulieren.

Wir unterscheiden zwischen Architekturen, die Eigenschaften des Anwendungssystems, des zu entwickelnden Softwaresystems oder dessen Umgebung festlegen. Weiterhin führen wir Produktarten ein, um einer Architektur die Rolle der jeweils feinsten Architektur (einer Entwicklung) zuordnen zu können, das heißt der Architektur, welche die stärksten Anforderungen an die Realisierung stellt. So dient etwa die feinste Umgebungsarchitektur dazu, die stärksten zulässigen Umgebungsannahmen festzulegen.

Definition 6.8 (Architekturen)

Wir unterscheiden Anwendungs-, Software- und Umgebungsarchitekturen.

Eine (endliche) Menge X von Entwicklungsprodukten ist

- eine *Anwendungsarchitektur*, falls gilt

$$\exists! i \in \mathbb{N}_+ . \forall sys, sys' \in [\![X]\!] .$$
$$i \in \left[1,...,(n_{snw})_{sys} \right] \wedge$$
$$nw_real\left((snw_i)_{sys} \right) = nw_real\left((snw_i)_{sys'} \right)$$

- eine *Softwarearchitektur*, falls, analog zu oben (nur mit *dnw* an Stelle von *snw*), gilt

$$\exists! i \in \mathbb{N}_+ . \forall sys, sys' \in [\![X]\!] .$$
$$i \in \left[1,...,(n_{snw})_{sys} \right] \wedge$$
$$nw_real\left((dnw_i)_{sys} \right) = nw_real\left((dnw_i)_{sys'} \right)$$

- eine *Umgebungsarchitektur*, falls, analog zu oben (nur mit *enw* an Stelle von *snw*) gilt

$$\exists! i \in \mathbb{N}_+ . \forall sys, sys' \in [\![X]\!] .$$
$$i \in \left[1,...,(n_{snw})_{sys} \right] \wedge$$
$$nw_real\left((enw_i)_{sys} \right) = nw_real\left((enw_i)_{sys'} \right)$$

wobei

$$nw_real : TNW(CH) \rightarrow \wp(NW(CH))$$

die in Definition 2.21 gegebene Abbildung sei und $\exists! i$ bedeute, dass *genau ein i* existiere.

□

Mit obigen Definitionen gilt, dass die durch eine Architektur festgelegte Menge von Entwicklungssystemen bezüglich (genau) eines Spezifikationsnetzwerkes, beziehungsweise bezüglich des Entwicklungs- oder Umgebungsanteils eines solchen, übereinstimmen. Für andere, davon unabhängige Systemmodellelemente, macht eine Architektur keine Aussagen. Eine Architektur beschränkt sich somit auf die Festlegung eines spezifikatorischen Netzwerkes eines Entwicklungssystems.

Zu beachten ist, dass wir mit obiger Definition nicht fordern, dass alle Eigenschaften eines Spezifikationsnetzwerkes durch eine Architektur eindeutig festgelegt werden. Mit dieser „offenen“ Definition stellt beispielsweise eine Produktmenge, mit der zwar das Interaktionsverhalten-, nicht aber Zustandsraum und Ausführungsfolgen von Komponenten definiert werden, auch eine Architektur dar. Dies ist beispielsweise sinnvoll im Fall von reinen Steuerungssystemen, durch die kein fachlicher (Daten-)Zustandsraum zu realisieren ist, so dass mit einem Spezifikationsnetzwerk kein Bezug zwischen Datenzustand und Interaktionsverhalten herzustellen ist.

Durch die folgende Normalform von Architekturen geben wir eine minimale Menge von Entwicklungsprodukten an, durch die eine Architektur formuliert werden kann.

Definition 6.9 (Normalformen von Architekturen)

Eine *Anwendungsarchitektur in Normalform* besteht aus

- einer exakten Spezifikation $(i, X)_{K_snw}$ der Komponentenmenge eines Spezifikationsnetzwerkes, sowie
- einer X-indizierten Menge

  $\left\{ (k, I_k, O_k, R_k)_{K_behav} \middle| k \in X \right\}$

  von Komponenten-Interaktionsverhaltensspezifikationen gemäss Definition 5.14.
- einer X-indizierten Menge

  $\left\{ (k, t_k)_{tt} \middle| k \in X \right\}$

  von Topologieklassifikationen gemäss Definition 5.8
- einer Y indizierten Menge

  $\left\{ (c, N_c)_{ct} \middle| c \in Y \right\}$

  exakter Kanaltypisierungen gemäss Definition 5.2, wobei $Y =_{def} \bigcup_{k \in X} I_k \cup O_k$.

Die *Normalform einer Software-* beziehungsweise einer *Umgebungsarchitektur* bauen wir entsprechend auf, wobei diese anstelle von $(i, X)_{K_snw}$ eine Spezifikation $(i, X)_{K_dnw}$ beziehungsweise $(i, X)_{K_enw}$ umfassen.

□

Mit einer Normalform gemäss obiger Definition legen wir die Menge der Komponenten eines spezifikatorischen Netzwerkes snw_i, dnw_i, oder enw_i eindeutig fest. Zudem wird die syntaktische Schnittstelle und die Verhaltensfunktion jeder dieser Komponenten eindeutig bestimmt. Zu beachten ist, dass mit der eindeutigen Festlegung der Verhaltensfunktion einer Komponente nicht notwendigerweise ein deterministisches Verhalten festgelegt ist, sondern nur der Grad des Nichtdeterminismus/der Grad der Unterspezifikation. Weiterhin ist zu beachten, dass die mit einer Normalform gegebenen Eindeutigkeiten auf Ebene der spezifikatorischen Komponenten, die Eindeutigkeit der durch ein snw_i, dnw_i, oder enw_i gestellten Anforderungen an das *nw*, *dnw* oder *enw* impliziert. Die Umkehrung gilt jedoch nicht. So sind etwa Unterschiede von topologischen *behavior*-Komponenten möglich.

Während wir eine Produktmenge, die eine Architektur in obigem Sinne darstellt, dadurch charakterisieren, dass in dieser Produktmenge alle, mit einer bestimmten spezifikatorischen Sicht an das Realisierungsnetzwerk gestellten Anforderungen

erfasst sind, halten wir darüber hinaus zwei spezifische Formen von Architekturen für methodisch bedeutsam, nämlich die sogenannten

- *Schnittstellen*- und
- *Realisierungsarchitekturen*

des zu entwickelnden Softwaresystems.

Unter Ersteren verstehen wir Architekturen, welche eine spezifikatorische Sicht des Softwaresystems angeben, die ausschließlich aus interface-Komponenten, das heißt Komponenten der topologischen Klasse *interface* (vergleiche Definition 2.17), aufgebaut sind. Analog dazu gibt eine Realisierungsarchitektur eine spezifikatorische Sicht an, die ausschließlich als *real* (vergleiche Definition 2.17) klassifizierte Komponenten verwendet.

Wie in den Abschnitten 2.5.3 und 2.5.4 erläutert, stellen wir mit Komponenten der topologischen Klasse *interface* modulare, zustands- und verhaltenskapselnde Einheiten dar, die als solche in einer Realisierung identifizierbar sein müssen. Damit sind interface-Komponenten ein Mittel, um topologische Anforderungen an eine komponentenbasierte Realisierung zu formulieren, ohne die Topologie der Realisierung eindeutig festlegen zu müssen. Im Forward-Engineering kann durch Angabe einer Schnittstellen-Architektur die Erfüllung von Qualitätskriterien, wie Erweiter- und Änderbarkeit der Softwarelösung sowie die Wiederverwendbarkeit einzelner Komponenten der Realisierung sichergestellt werden. Zudem sind Schnittstellenarchitekturen zentral bei der Verfolgung einer inkrementellen und/oder parallelen Entwicklungsstrategie. Im Rahmen einer inkrementellen Entwicklung ist beispielsweise folgendes Vorgehen vorstellbar:

1. Schritt: Schnittstellenarchitektur definieren.
2. Schritt: Schnittstellenarchitektur in Inkremente partitionieren
3. Schritt: einzelne Inkremente entwickeln und realisieren

Indem durch die Schnittstellenarchitektur die Schnittstellen zwischen den Inkrementen festgelegt sind, ist sichergestellt, dass die Inkremente zu einem sinnvollen Ganzen komponiert werden können.

Bei der Entwicklung von Ad-hoc-Systemen (vergleiche etwa [Sal02]) kann in Form einer Schnittstellen-Architektur die sogenannte Dienst-Architektur eines Systems angegeben werden. Die feinste Dienstarchitektur legt damit die Menge der zulässigen *Deployments* fest (vergleiche [Sal02] und [SS01]).

Im Unterschied zu interface-Komponenten, repräsentieren jede, mit *real* (vergleiche Definition 2.17) topologisch klassifizierte Komponente (eines Spezifikationsnetzwerkes), eine Komponente des Realisierungsnetzwerkes, die eine übereinstimmende syntaktische Schnittstelle und ein konsistentes beobachtbares Verhalten aufweist. Somit legt eine Realisierungsarchitektur die Topologie der Realisierung des Softwaresystems eindeutig fest. Eine feinste Realisierungsarchitektur des Softwaresystems stellt zudem die stärksten Anforderungen an das beobachtbare Verhalten der Komponenten einer Realisierung, das heißt eines Entwicklungsnetzwerkes *dnw*.

In einer hierarchisch strukturierten Spezifikation eines Softwaresystems, wie sie etwa in Form der *Erzeugnisstruktur* des V-Modells gefordert wird, ist die feinste Realisierungsarchitektur aus den spezifikatorischen Einheiten der niedrigsten Hierarchieebene, im V-Modell [BDI97] sind dies *Software-Module*, komponiert.

Grundsätzlich eignet sich eine Realisierungsarchitektur aufgrund der topologischen Äquivalenz von Spezifikation und Realisierung zur Dokumentation der Realisierung/Implementierung. Im Unterschied zur Implementierung, zum Beispiel in Form von Quellcode, haben wir mit der Realisierungsarchitektur jedoch die Möglichkeit, Systemeigenschaften auf implementierungsunabhängige Weise, das heißt unter Verwendung von Modellierungskonzepten, die sich von denen der Implementierungssprache unterscheiden, anzugeben.

Definition 6.10 (Realisierungs- und Schnittstellenarchitektur des Softwaresystems)

Eine Produktmenge X ist eine *Realisierungsarchitektur des Softwaresystems*, falls

- X eine Softwarearchitektur gemäss Definition 6.8 ist, und
- das spezifizierte Entwicklungsnetzwerk ausschließlich aus Komponenten besteht, die topologisch als *real* Komponenten klassifiziert sind (vergleiche Definition 2.17),

das heißt, es gilt

$$\exists! i \in \mathbb{N}_{+}. \forall sys, sys' \in [\![X]\!].$$
$$i \in \left[1, \ldots, (n_{snw})_{sys}\right] \wedge$$
$$nw_real\left((dnw_i)_{sys}\right) = nw_real\left((dnw_i)_{sys'}\right) \wedge$$
$$\left(\forall c \in (dnw_i)_{sys}.tclass(c) = real\right)$$

Analog dazu, ist eine Produktmenge X eine *Schnittstellenarchitektur des Softwaresystems*, falls

- X eine Softwarearchitektur gemäss Definition 6.8 ist, und
- das spezifizierte Entwicklungsnetzwerk ausschließlich aus Komponenten besteht, die als *interface* Komponenten topologisch klassifiziert sind,

das heißt, es gilt

$$\exists! i \in \mathbb{N}_{+}. \forall sys, sys' \in [\![X]\!].$$
$$i \in \left[1, \ldots, (n_{snw})_{sys}\right] \wedge$$
$$nw_real\left((dnw_i)_{sys}\right) = nw_real\left((dnw_i)_{sys'}\right) \wedge$$
$$\left(\forall c \in (dnw_i)_{sys}.tclass(c) = interface\right)$$

□

Um einer Realisierungsarchitektur die Rolle der *feinsten* Realisierungsarchitektur (des zu entwickelnden Softwaresystems) einer Entwicklung zuordnen zu können,

führen wir die gleichnamige Produktart ein. Analog dazu, führen wir die Produktart der feinsten Umgebungsarchitektur ein, durch welche die stärksten (zulässigen) Umgebungsannahmen formuliert werden können. Diese Produktart lässt sich sinnvoll mit einer Anforderungsspezifikation gemäss Definition 6.7 oder auch nur einer Spezifikation der fachlichen Aufgabenstellung kombinieren, um auf diese Weise neben den fachlichen und den Nutzungsanforderungen an das Softwaresystem, auch die zulässigen Umgebungsannahmen zu erfassen.

Definition 6.11 (Feinste Software- und Umgebungsarchitektur)

Die Produktart zur Auszeichnung der *feinsten Softwarearchitektur* einer Entwicklung bezeichnen wir *dnw!*, deren Semantik wir wie folgt definieren:

$$\begin{aligned}[\![(X)_{dnw!}]\!] =_{def} \{ & sys \in SYS \mid sys \in [\![X]\!] \wedge \\ & \forall i \in \left[1,\ldots,(n_{snw})_{sys}\right]. \\ & \quad specified_dnw(X,i) \Rightarrow \\ & \quad \forall j \in \left[1,\ldots,(n_{snw})_{sys}\right].nw_real\left((dnw_i)_{sys}\right) \subseteq nw_real\left((dnw_j)_{sys}\right) \}\end{aligned}$$

mit

$$specified_dnw : (\wp(EP) \times \mathbb{N}_+) \rightarrow \mathbb{B},$$

$$\begin{aligned}& specified_dnw(X,i) \Leftrightarrow_{def} \\ & \quad \forall sys, sys' \in [\![X]\!].nw_real\left((dnw_i)_{sys}\right) = nw_real\left((dnw_i)_{sys'}\right)\end{aligned}$$

für alle $X \in \wp(EP), i \in \mathbb{N}_+$. *specified_dnw(X,i)* ist genau dann wahr, wenn durch die Produktmenge *X* die, durch das Netzwerk dnw_i formulierten Anforderungen an *dnw*, für jedes System $sys \in [\![X]\!]$, eindeutig festgelegt sind.

Analog dazu definieren wir die Produktart der feinsten Umgebungsarchitektur *enw!* wie folgt:

$$\begin{aligned}[\![(X)_{enw!}]\!] =_{def} \{ & sys \in SYS \mid sys \in [\![X]\!] \wedge \\ & \forall i \in \left[1,\ldots,(n_{snw})_{sys}\right]. \\ & \quad specified_enw(X,i) \Rightarrow \\ & \qquad \forall j \in \left[1,\ldots,(n_{snw})_{sys}\right]. \\ & \qquad nw_real\left((enw_i)_{sys}\right) \subseteq nw_real\left((enw_j)_{sys}\right) \}\end{aligned}$$

mit

$$\text{specified_enw} : (\wp(\text{EP}) \times \mathbb{N}_+) \rightarrow \mathbb{B},$$

$$\text{specified_enw}(X,i) \Leftrightarrow_{\text{def}}$$
$$\forall sys, sys' \in [\![X]\!].\text{nw_real}\big((\text{enw}_i)_{sys}\big) = \text{nw_real}\big((\text{enw}_i)_{sys'}\big)$$

für alle $X \in \wp(\text{EP}), i \in \mathbb{N}_+$.

□

Im Unterschied zur feinsten Softwarearchitektur, zeichnen wir die feinste Schnittstellen-Architektur nicht in Bezug zu allen Entwicklungsnetzwerken aus, sondern nur in Bezug zu jenen, die ausschließlich aus Komponenten bestehen, welche topologisch als *interface* Komponenten klassifiziert sind:

Definition 6.12 (Feinste Schnittstellenarchitektur)

Die Produktart zur Auszeichnung der *feinsten Schnittstellenarchitektur des Softwaresystems* einer Entwicklung bezeichnen wir *interface_dnw!*, deren Semantik wir wie folgt definieren:

$$\begin{aligned}\big[\!\big[(X)_{\text{interface_dnw!}}\big]\!\big] =_{\text{def}} \{sys \in \text{SYS} \mid\, & sys \in [\![X]\!] \wedge \\ & \forall i \in \big[1,\ldots,(n_{snw})_{sys}\big]. \\ & \text{specified_dnw}(X,i) \Rightarrow \\ & \forall j \in \big[1,\ldots,(n_{snw})_{sys}\big]. \\ & \Big(\big(\forall c \in (\text{dnw}_j)_{sys}.\text{tclass}(c) = \text{interface}\big) \Rightarrow \\ & \text{nw_real}\big((\text{dnw}_i)_{sys}\big) \subseteq \text{nw_real}\big((\text{dnw}_j)_{sys}\big)\Big)\Big\}\end{aligned}$$

mit specified_dnw wie in Definition 6.11 gegeben.

□

Eine feinste Schnittstellenarchitektur gemäss obiger Definition ist ein Mittel, um *alle* in einer Entwicklung festgelegten und in der Realisierung *einzuhaltenden Schnittstellen* anzugeben. Gegenüber einer feinsten Schnittstellenarchitektur ist eine topologische Verfeinerung nur durch Festlegung von Realisierungskomponenten, das heißt mit *real* topologischen klassifizierten Komponenten, möglich. Daher stellt eine feinste Schnittstellenarchitektur eine geeignete Vorgabe für die Entwicklung einer Realisierungsarchitektur (entsprechend Definition 6.10) dar, etwa im Rahmen des Feinentwurfs oder der Implementierungsphase eines Top-Down-Vorgehens.

Für unsere Beispielentwicklung des Bibliothekssystems geben wir in Beispiel 6.7 die feinste Umgebungsarchitektur an, wodurch wir die Umgebungsannahmen festlegen.

Beispiel 6.7 (Umgebungsarchitektur)

In unserem Bibliotheksbeispiel gehen wir davon aus, dass die relevanten Umgebungsaspekte des Bibliothekssoftwaresystems durch die beiden, die Anwenderrollen repräsentierenden Komponenten *Leser* und *BibAngest* sowie die Komponente *E-Mailer* gegeben sind. Während wir mit *Leser* und *BibAngest* lediglich die fachliche Einbettung charakterisieren, ohne damit Annahmen über das Verhalten der beiden Arten von Nutzern zu verbinden, stellt die Komponente E-Mailer eine „technische" Komponente dar, von der wir folgendes Verhalten fordern:

Auf jede Anforderung, eine E-Mail an eine bestimmte E-Mail-Adresse mit einem bestimmten (ASCII-)Text zu versenden, reagiert die Komponente mit der Ausgabe (mindestens) einer entsprechenden TCP-IP-konformen Nachricht (in das Internet). Mit der (Interface-)Komponente *E-Mailer* erfassen wir somit die, beispielsweise durch ein E-Mail-Programm oder ein Betriebssystem bereitgestellte Funktionalität des E-Mail-Versandes. Da wir *E-Mailer* als Teil der Umgebung festlegen, formulieren wir damit die Annahme, dass die Umgebung des Bibliothekssystems eine entsprechende Funktionalität zur Verfügung stellt.

Die (feinste) Umgebungsarchitektur geben wir anhand des mit 1 indizierten Spezifikationsnetzwerkes, snw_1, an, für das wir bereits in Beispiel 5.2 Eigenschaften der statischen Struktur und in Beispiel 5.3 anhand eines Reaktionsszenarios Verhalteneigenschaften festgelegt haben. Abbildung 6.6 gibt einen Überblick über die statische Struktur des Netzwerkes.

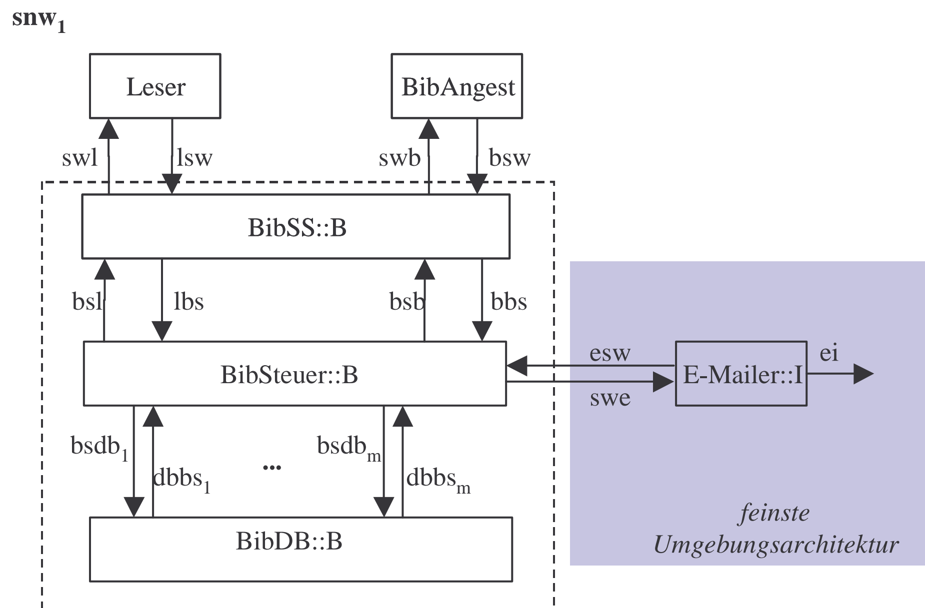


Abbildung 6.6:Statische Struktur des Spezifikationsnetzwerkes snw_1 der Bibliothekssystem-Entwicklung.

Mit der Produktmenge

$$
\begin{aligned}
\text{ENVARCH} =_{\text{def}} \{ & (\text{MailMsg} \cup \text{TcpipMsg})_{N\supseteq}, \\
& (1, \{\text{Leser}, \text{BibAngest}, \text{E-Mailer}\})_{K_enw}, \\
& (\text{E-Mailer}, \{\text{swe}\}, \{\text{esw}, \text{ei}\}, R_{EM})_{K_behav}, \\
& (\text{Leser}, \{\text{swl}\}, \{\text{lsw}\}, R_L)_{K_behav}, \\
& (\text{BibAngest}, \{\text{swb}\}, \{\text{bsw}\}, R_B)_{K_behav}, \\
& (\text{E-Mailer}, \text{interface})_{tt}, \\
& (\text{Leser}, \text{behavior})_{tt}, \\
& (\text{BibAngest}, \text{behavior})_{tt}, \\
& \text{CHTYPE}\}
\end{aligned}
$$

definieren wir die feinste Umgebungsarchitektur (gemäss Definition 6.11) unseres Bibliotheksbeispiels durch

$$(\text{ENVARCH})_{enw!}$$

wobei

$$R_L =_{\text{def}} \text{true}$$
$$R_B =_{\text{def}} \text{true}$$

und für alle $\varphi \in \overrightarrow{\{\text{swe}, \text{esw}, \text{ei}\}}$ gelte:

$$
\begin{aligned}
& R_{EM}(\varphi) \Leftrightarrow_{\text{def}} \\
& \forall n \in \mathbb{N}_+, a \in \text{MailAdr}, t \in \text{ASCIIString}. \\
& (in(\text{sendmail}(a, t), ((\varphi @ n)(\text{swe}))) \Rightarrow \\
& \qquad (\exists n' \in \mathbb{N}_+ . n' > n \wedge \\
& \qquad in(\text{tcpip}(\text{mailserver}(a), \text{mcontent}(a, t)), \\
& \qquad\quad ((\varphi @ n')(\text{ei}))))) \\
& \wedge \\
& ((\#(\{\text{sendmail}(a, t)\} \copyright \varphi|_{swe}) < \\
& \#(\{\text{tcpip}(\text{mailserver}(a), \text{mcontent}(a, t))\} \copyright \varphi|_{ei})))
\end{aligned}
$$

wobei

$$\text{sendmail} : \text{MailAdr} \times \text{ASCIIString} \rightarrow \text{MailMsg}$$
$$\text{mailserver} : \text{MailAdr} \rightarrow \text{NodeAdr}$$
$$\text{mcontent} : \text{MailAdr} \times \text{ASCIIString} \rightarrow \text{TcpipData}$$
$$\text{tcpip} : \text{NodeAdr} \times \text{TcpipData} \rightarrow \text{TcpipMsg}$$

entsprechende Konstruktorfunktionen seien. Weiterhin sei CHTYPE eine geeignete {swe, esw, ei, swl, lsw, swb, bsw}-indizierte Menge exakter Kanaltypisierungen.

□

Mit der in Beispiel 6.8 gegebenen feinsten Schnittstellenarchitektur unserer exemplarischen Bibliothekssoftware-Entwicklung geben wir eine, für die Realisierung/Implementierung bindende Dekomposition/Modularisierung des zu entwickelnden Softwaresystems an. Wir wenden hierbei das Prinzip der „Trennung unabhängiger Aspekte" an, und schaffen damit die Voraussetzung für die Erfüllung von Qualitätsanforderungen, wie Änderbarkeit sowie für eine inkrementelle und/oder parallele Entwicklung.

Beispiel 6.8 (feinste Schnittstellenarchitektur)

Für Informationssysteme hat sich die Einhaltung einer 3-Schichten-Architektur bewährt, in der Benutzer-, Steuerungs- und Datenhaltungsaufgaben in aufeinander aufbauenden, getrennten Schichten realisiert sind. In Beispiel 5.1 beschreiben wir die Gesamtfunktionalität unseres Bibliothekssystems bereits anhand einer solchen Struktur, die aus den drei, die Schichten repräsentierenden Komponenten *BibSS*, *BibSteuer* und *BibDB* besteht. Diese sind jedoch topologisch als behavior-Komponenten klassifiziert, so dass die mit dieser Architektursicht gegebene Strukturierung für die Realisierung/Implementierung nicht bindend ist.

In der unten gegebenen Sicht, welche die feinste Schnittstellenarchitektur (gemäss Definition 6.12) unserer Beispielentwicklung darstelle, legen wir die Trennung in drei Schichten für die Realisierung bindend fest, indem wir interface-Komponenten zur Repräsentation der Schichten verwenden. Die Schicht der Benutzerschnittstellen repräsentieren wir durch die Komponente *BibSSI*, die Datenhaltungsschicht durch die Komponente *BibDBI*. Die Kontrollschicht, welche die Steuerung der Geschäftsprozesse zur Aufgabe hat, erfassen wir als Netzwerk von vier Komponenten.

Gemäss Abschnitt 1.7 hat unser Bibliothekssystem die Vormerkung, Rückgabe und Ausleihe von Medien zu unterstützen. Wie die jeweiligen Abläufe zu steuern sind, ist unabhängig voneinander. Daher fordern wir (mittels der Schnittstellenarchitektur), dass die jeweilige Funktionalität an separierten Schnittstellen (das heißt, auf unterschiedlichen Kanälen) zur Verfügung gestellt wird. Auf diese Weise erreichen wir eine klare Trennung unabhängiger Aspekte und damit die in der informellen Aufgabenstellung (vergleiche Abschnitt 1.7) geforderte Änder- und Erweiterbarkeit des Softwaresystems bezüglich der zu unterstützenden Geschäftsprozesse. Indem wir jeweils eine separate (interface-)Komponente, *V_Ctrl*, *A_Ctrl* beziehungsweise *R_Ctrl*, in der Schnittstellenarchitektur angeben, machen wir zudem die Unabhängigkeit der Funktionalitäten transparent. Um zwischen diesen Komponenten und der Benutzerschnittstellenschicht, an der Geschäftsprozesse angestoßen werden, eine Verbindung herzustellen, führen wir zudem die Komponente *Ctrl_Connect* ein.

Abbildung 6.7 gibt einen Überblick der statischen Struktur des Spezifikationsnetzwerkes.

Zu beachten ist, dass die, durch die interface-Komponenten festgelegte Dekomposition des Bibliothekssystems noch Freiheitsgrade für die Wahl der Topologie der Realisierung lässt. Beispielsweise ist damit nicht ausgeschlossen, dass, etwa aus Performanzgründen oder um existierende COTS- oder legacy-Komponenten einzubinden, mehrere interface-Komponenten durch *eine* Komponente des Realisierungsnetzwerkes realisiert werden. In Beispiel 6.10 fassen wir etwa *V_Ctrl* und *A_Ctrl* in der Komponente *VA_Ctrl* zusammen. Ebenso ist die Verfeinerung einer interface-Komponente zu einem, diese Komponente realisierenden Netzwerk zulässig. In Beispiel 6.10 „verfeinern" wir die Komponente *BibSSI* zu dem aus den Komponenten *LSS* und *BASS* bestehenden Netzwerk.

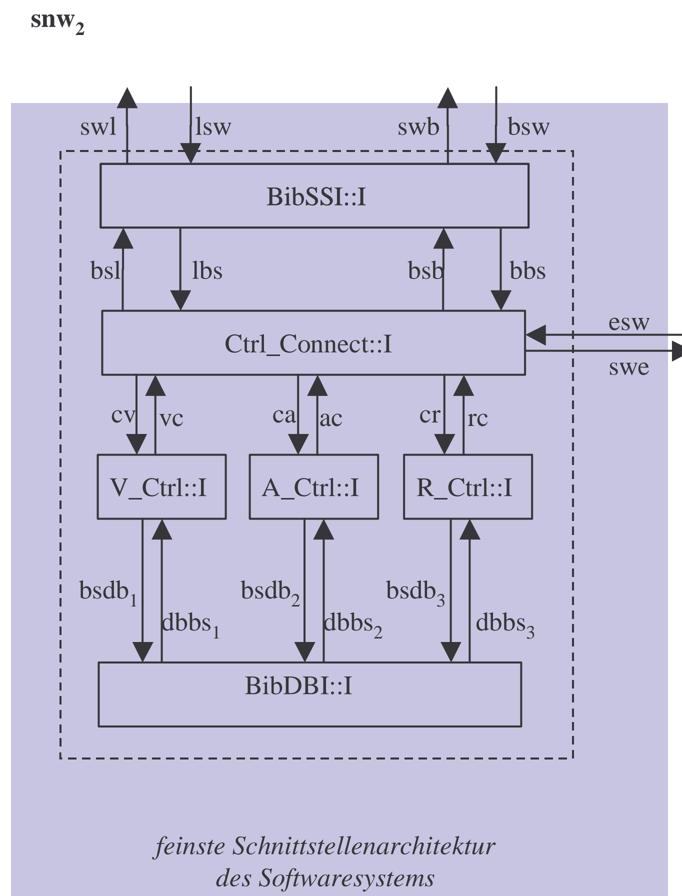


Abbildung 6.7: Spezifikationsnetzwerk snw2 zur Angabe der feinsten Schnittstellenarchitektur des Bibliothekssystems.

Die Schnittstellenarchitektur geben wir durch die Produktmenge

SSARCH

an:

$(\text{SSARCH})_{\text{interface_dnw!}}$

SSARCH definieren wir wie folgt:

$$
\begin{aligned}
&\text{SSARCH} =_{\text{def}} \\
&\{(2, \{\text{BibSSI}, \text{Ctrl_Connect}, \text{V_Ctrl}, \text{A_Ctrl}, \text{R_Ctrl}, \text{BibDBI}\})_{\text{K_dnw}}, \\
&(\text{BibSSI}, I_{\text{BibSSI}}, O_{\text{BibSSI}}, R_{\text{BibSSI}})_{\text{K_behav}}, \\
&(\text{Ctrl_Connect}, I_{\text{CtrlCnct}}, O_{\text{CtrlCnct}}, R_{\text{CtrlCnct}})_{\text{K_behav}}, \\
&(\text{V_Ctrl}, S_{\text{VC}}, I_{\text{VCtrl}}, O_{\text{VCtrl}}, \delta_{\text{VC}}, \text{init}_{\text{VC}})_{\text{stm}} \\
&(\text{A_Ctrl}, I_{\text{ACtrl}}, O_{\text{ACtrl}}, R_{\text{ACtrl}})_{\text{K_behav}}, \\
&(\text{R_Ctrl}, I_{\text{RCtrl}}, O_{\text{RCtrl}}, R_{\text{RCtrl}})_{\text{K_behav}}, \\
&(\text{BibDBI}, I_{\text{BibDBI}}, O_{\text{BibDBI}}, R_{\text{BibDBI}})_{\text{K_behav}}, \\
&(\text{BibSSI}, \text{interface})_{\text{tt}}, (\text{Ctrl_Connect}, \text{interface})_{\text{tt}}, \\
&(\text{V_Ctrl}, \text{interface})_{\text{tt}}, (\text{A_Ctrl}, \text{interface})_{\text{tt}}, (\text{R_Ctrl}, \text{interface})_{\text{tt}}, \\
&(\text{BibDBI}, \text{interface})_{\text{tt}}, \\
&\text{CHTYPE}'\}
\end{aligned}
$$

Hierbei seien die mit den Komponentenbezeichnern indizierten Mengen der Ein- und Ausgabekanäle, I beziehungsweise O, der jeweiligen Komponenten wie in Abbildung 6.7 dargestellt gegeben.

Das Verhalten der einzelnen Komponenten charakterisieren wir wie folgt:

BibSSI.

Die Komponente *BibSSI* dient dazu, Benutzern des Softwaresystems geeignete Dialoge/Masken zur Verfügung zu stellen, um die Vormerkung, Rückgabe und Ausleihe von Medien anzustoßen (vergleiche Beispiel 6.5). Das Verhalten von *BibSSI* entspricht folgendem Schema:

> Nach Auswahl des gewünschten Geschäftsprozesses und der Eingabe der benötigten Informationen, beispielsweise welches Medium für welchen Leser vorzumerken ist, wird eine Nachricht an die Steuerungsschicht, das heißt die Komponente *Ctrl_Connect*, gesendet, um so die (Softwaresystem-internen) Aktionen des jeweiligen Geschäftsprozesses anzustoßen.

Wir definieren daher folgende Nachrichtenmengen:

$$V =_{\text{def}} \{x \mid \exists m \in (A_{\text{sys}})_{\text{id}_{\text{Medium}}}, l \in (A_{\text{sys}})_{\text{id}_{\text{Leser}}} . x = \text{vormerkungswunsch}(m, l)\}$$

$$A =_{\text{def}} \{x \mid \exists m \in (A_{\text{sys}})_{\text{id}_{\text{Medium}}} . x = \text{ausleihe}(m)\}$$

$$R =_{\text{def}} \{x \mid \exists m \in (A_{\text{sys}})_{\text{id}_{\text{Medium}}} . x = \text{rückgabe}(m)\}$$

für ein System $\text{sys} \in \text{SYS}$, wobei wir davon ausgehen, dass $\text{id}_{\text{Medium}}$ und id_{Leser} Identifikatorsorten der Entitätssignatur $(\Sigma_{\text{fe}})_{\text{sys}}$ seien und

$$\text{vormerkungswunsch} : (A_{sys})_{id_{Medium}} \times (A_{sys})_{id_{Leser}} \rightarrow N_{sys}$$
$$\text{ausleihe} : (A_{sys})_{id_{Medium}} \rightarrow N_{sys}$$
$$\text{rückgabe} : (A_{sys})_{id_{Medium}} \rightarrow N_{sys}$$

Konstruktorfunktionen von Nachrichten seien.

Entsprechend der informellen Anforderungen aus Abschnitt 1.7, können sowohl Leser als auch Bibliotheksangestellte Vormerkungen anstoßen, während die Rückgabe und Ausleihe von Medien nur durch Bibliotheksangestellte möglich sein soll. Dies spiegelt sich in den folgenden Kanaltypisierungen wider:

$$\{(lbs, V)_{ctype}, (bbs, V \cup A \cup R)_{ctype},$$
$$(cv, V)_{ctype}, (ca, A)_{ctype}, (cr, R)_{ctype}\} \subseteq CHTYPE'$$

wobei CHTYPE´ für alle weiteren Kanäle der Architektur geeignete Typisierungen enthalte.

Ctrl_Connect.

Die Aufgabe der Komponente *Ctrl_Connect* besteht darin, die Eingaben aus der Benutzerschicht an die zugehörige Steuerungskomponente weiterzuleiten, das heißt Vormerkungsnachrichten werden an V_Ctrl, Rückgabenachrichten an R_Ctrl und Ausleihenachrichten an A_Ctrl (reihenfolgeerhaltend) weitergeleitet. Entsprechend definieren wir das Verhalten von *Ctrl_Connect* durch folgendes Prädikat:

$$R_{Ctrl_Connect} : \overrightarrow{(I_{Ctrl_Connect} \cup O_{Ctrl_Connect})} \rightarrow \mathbb{B}$$

wobei für alle $\varphi \in \overrightarrow{(I_{Ctrl_Connect} \cup O_{Ctrl_Connect})}$ gelte:

$$\begin{aligned} & R_{Ctrl_Connect}(\varphi) \Leftrightarrow_{def} \\ & \quad \forall i \in \mathbb{N}_+ . \\ & \quad (\varphi @ (i+1))(cv) \in merge(V © \varphi @ i(lbs), V © \varphi @ i(bbs)) \wedge \\ & \quad (\varphi @ (i+1))(ca) = A © \varphi @ i(bbs) \wedge \\ & \quad (\varphi @ (i+1))(cr) = R © \varphi @ i(bbs) \end{aligned}$$

wobei *merge* die reihenfolgeerhaltende Mischfunktion zweier Sequenzen sei.

V_Ctrl, A_Ctrl, R_Ctrl.

V_Ctrl, *A_Ctrl* und *R_Ctrl* steuern, wie erwähnt, die (Bearbeitung von) Vormerkung-, Ausleihe- beziehungsweise Rückgabe von Medien.

In Beispiel 6.9 gehen wir exemplarisch auf die Komponente *V_Ctrl* ein, deren Verhalten wir anhand einer Automatenspezifikation angeben. Für *A_Ctrl* und *R_Ctrl* nehmen wir geeignete Verhaltensspezifikationen als gegeben an.

BibDBI.

Das mit R_{BibDBI} charakterisierte Verhalten der Datenbankkomponente sei konform zu der in Beispiel 5.5 in Form eines Automaten gegebenen Verhaltensspezifikation.

Topologie-Klassifikation.

Wie für eine Schnittstellenarchitektur gefordert, klassifizieren wir alle Komponenten des Netzwerkes (topologisch) als interface-Komponenten.

□

Mit der Spezifikation der Datenbank-Komponente aus Beispiel 5.5, sowie mit obiger Spezifikation der Komponente *Ctrl-Connect*, zusammen mit der folgenden Spezifikation der mit der Schnittstellenarchitektur aus obigem Beispiel eingeführten Komponente *V_Ctrl*, geben wir einen „Durchstich" unseres Bibliothekssystems anhand der Aufgabe der Vormerkungsbearbeitung an.

Beispiel 6.9 (Automatenspezifikation der Komponente V_Ctrl)

Das mit folgender Automatenspezifikation festgelegte Interaktionsverhalten der Komponente *V_Ctrl*, deren Aufgabe die Steuerung der Bearbeitung von Vormerkungswünschen ist, wird zum einen bestimmt durch die in den zugehörigen Geschäftsprozessspezifikationen (vergleiche Beispiel 4.1 bis Beispiel 4.3) festgelegten Prozessereignisse und deren kausale Abhängigkeiten. Zum anderen fließt mit ein, dass, wie in Beispiel 5.5 spezifiziert, der gesamte fachliche Zustandsraum durch die Datenbank-Komponente unseres Bibliothekssystems realisiert wird, und somit zustandsbezogene (Geschäfts-)Prozessereignisse durch entsprechende Interaktion mit der Datenbank-Komponente zu realisieren sind. So bestimmen die Markierungen der Prozessereignisse die Art der auszutauschenden Nachrichten, die kausalen Abhängigkeiten zwischen den Ereignissen bestimmen die Folge der Interaktionen. Auf diese Weise illustriert das Beispiel, wie aufgrund der, durch das Systemmodell festgelegten, klaren Zusammenhänge zwischen den Prozessereignissen und Ausführungsfolgen von Komponentennetzwerken (vergleiche Definition 2.28) eine transparente schrittweise Entwicklung, in diesem Fall von der Geschäftsprozessspezifikation über die Festlegung der Zustandsraum-Realisierung bis hin zur Verhaltensspezifikation der Steuerungskomponente, möglich wird.

$$(V_Ctrl, S_{VC}, I_{VC}, O_{VC}, \delta_{VC}, init_{VC})_{stm}$$

mit

$$I_{VC} =_{def} \{cv, dbbs_l\}, O_{VC} =_{def} \{vc, bsdb_l\},$$

$$S_{VC} =_{def} V^* \times (A_{sys})_{id_{Medium}} \times \{wait, requ1, requ2, next\}$$

wobei die Nachrichtenmenge *V* wie in Beispiel 6.8 definiert sei. Das erste Element eines 3-Tupels aus S_{VC} verwenden wir als Puffer der Eingaben von *Ctrl_Connect* an *V_Ctrl* der Form

vormerkungswunsch(m,l)

durch welche eine entsprechende Vormerkungsbearbeitung angestoßen wird. Das zweite Element eines Zustands aus *V_Ctrl* dient als Hilfsvariable für die passende Parametrisierung von Nachrichten, die von *V_Ctrl* an die Datenbank (im Rahmen einer Vormerkungsbearbeitung) zu senden sind. Das dritte Element steht für die unterschiedlichen Kontrollzustände, die im Laufe der Bearbeitung von Vormerkungen eingenommen werden.

Damit ergibt sich die Menge der Initialzustände wie folgt:

$$\text{init}_{VC} =_{def} \{ s \in S_{VC} \mid \Pi_1(s) = \varepsilon \wedge \Pi_3(s) = \text{wait} \}.$$

Durch die unten spezifizierte Transitionsfunktion legen wir folgendes Verhaltensschema fest:

> Vormerkungswünsche werden in der Reihenfolge der Eingaben (von *Ctrl_Connect*) streng sequentiell bearbeitet, entsprechend den Schritten und Alternativen, wie sie in den zugehörigen Geschäftsprozessspezifikationen (vergleiche Beispiel 4.1 bis Beispiel 4.3) festgelegt sind. Hierzu sind Vormerkungswünsche, die während der Bearbeitung eines vorangegangenen Wunsches eintreffen, zu puffern. Welche Interaktionen zur Realisierung der Geschäftsprozessereignisse notwendig sind, ergibt sich aus der in Beispiel 5.5 getroffenen Festlegung, dass der gesamte fachliche Zustandsraum durch die Datenbank-Komponente realisiert wird, sowie durch die, in der Automatenspezifikation aus Beispiel 5.5 definierten Zusammenhänge zwischen Datenbank-Eingaben und dem realisierten, fachlichen Zustandsraum.

Die Eigenschaften der Transitionsfunktion definieren wir ausschnittsweise für eine der möglichen Alternativen einer Vormerkungsbearbeitung (vergleiche die genannten Geschäftsprozessspezifikation) folgendermaßen:

$$\forall s, s' \in S_{VC}, \varphi \in (\vec{I_{VC}})^*, \psi \in (\vec{O_{VC}})^*,$$
$$m \in (A_{sys})_{id_{Medium}}, l \in (A_{sys})_{id_{Leser}}, x \in V^*.$$

$$(s \in \text{init}_{VC} \wedge \varphi(cv) = \varepsilon) \Rightarrow$$
$$\delta_{VC}(s, \varphi) = \{(s', \psi) \mid s = s'\}$$

$$\wedge$$

$$(s \in \text{init}_{VC} \wedge \varphi(cv) = \text{vormerkungswunsch}(m, l) \,\&\, x) \Rightarrow$$
$$\delta_{VC}(s, \varphi) = \{(s', \psi) \mid$$
$$(\Pi_1(s') = \Pi_1(s) \circ x) \wedge$$
$$(\Pi_2(s') = m) \wedge$$
$$(\Pi_3(s') = \text{requ1}) \wedge$$
$$\psi(bsdb_1) = \left\langle m.\text{vormerken}^*(l)^{A_{sys}} \right\rangle\}$$

$\wedge$

$$
\begin{aligned}
&(\Pi_3(s) = requ1 \wedge \Pi_2(s) = m) \Rightarrow \\
&\qquad \delta_{VC}(s,\varphi) = \{(s',\psi) | \\
&\qquad\qquad (\Pi_1(s') = \Pi_1(s) \circ \varphi(cv)) \wedge \\
&\qquad\qquad (\Pi_2(s') = m) \wedge \\
&\qquad\qquad (\Pi_3(s') = requ2) \wedge \\
&\qquad\qquad \psi(bsdb_1) = \left\langle m.eqstatus(verfügbar)^{A_{sys}} \right\rangle\}
\end{aligned}
$$

$\wedge$

$$
\begin{aligned}
&(\Pi_3(s) = requ2 \wedge \Pi_2(s) = m \wedge \\
&in\left(\left(m.eqstatus^{A_{sys}}(verfügbar), true\right), \varphi(dbbs_1)\right)\right) \Rightarrow \\
&\qquad \delta_{VC}(s,\varphi) = \{(s',\psi) | \\
&\qquad\qquad (\Pi_1(s') = \Pi_1(s) \circ \varphi(cv)) \wedge \\
&\qquad\qquad (\Pi_2(s') = m) \wedge \\
&\qquad\qquad (\Pi_3(s') = next) \wedge \\
&\qquad\qquad \psi(bsdb_1) = \left\langle m.zuteilen(\ )^{A_{sys}} \right\rangle\}
\end{aligned}
$$

$\wedge$

$$
\begin{aligned}
&(\Pi_3(s) = next \wedge \Pi_1(s) = \varepsilon \wedge \varphi(cv) = \varepsilon) \Rightarrow \\
&\qquad \delta_{VC}(s,\varphi) = \{(s',\psi) | (\Pi_1(s') = \varepsilon) \wedge (\Pi_3(s') = wait)\}
\end{aligned}
$$

$\wedge$

$$
\begin{aligned}
&(\Pi_3(s) = next \wedge \Pi_1(s) = vormerkungswunsch(m,1) \circ x) \Rightarrow \\
&\qquad \delta_{VC}(s,\varphi) = \{(s',\psi) | \\
&\qquad\qquad \Pi_1(s') = x \circ \varphi(cv) \wedge \\
&\qquad\qquad \Pi_2(s') = m \wedge \\
&\qquad\qquad (\Pi_3(s') = requ1) \wedge \\
&\qquad\qquad \psi(bsdb_1) = \left\langle m.vormerken^{*}(1)^{A_{sys}} \right\rangle\}
\end{aligned}
$$

$\wedge$

--

(der Geschäftsprozessspezifikation aus Beispiel 4.1, sowie der Datenbank-Spezifikation aus Beispiel 5.5) entsprechende Axiome für den Fall, dass *(m.eqstatus(verfügbar), false)* auf $dbbs_1$ empfangen wird, sowie Axiome für (technische) Fehlerfälle.

--

Zu beachten ist, dass obige Automatenspezifikation hochgradig nichtdeterministisch ist.

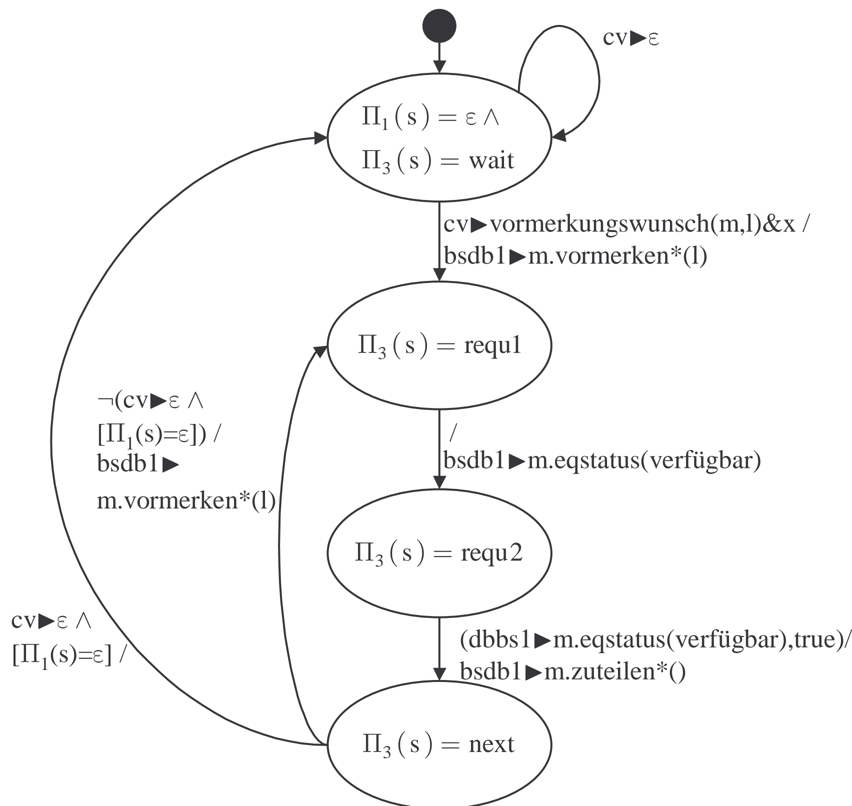


Abbildung 6.8: Automatendiagramm der Komponente *V_Ctrl*.

Die wesentliche Struktur des mit obiger Spezifikation gegebenen Automaten von *V_Ctrl* stellen wir in Abbildung 6.8 in Form eines Automatendiagramms dar. Hierbei zeigt sich, dass graphische Notationen in jedem Fall gut geeignet sind, um zentrale Charakteristika eines Automaten übersichtlich darzustellen. Jedoch stößt die graphische Darstellung unserer Meinung nach bei komplexer aufgebauten Zustandsklassen und bei Ein- und Ausgaben auf mehreren Kanälen schnell an ihre Grenzen.

□

Mit der im folgenden Beispiel gegebenen feinsten Realisierungsarchitektur unseres Bibliothekssystems, legen wir dessen Topologie eindeutig fest. Durch Bezugnahme auf die Schnittstellenarchitektur aus Beispiel 6.8 illustrieren wir zum einen, wie in unserem Ansatz Zusammenhänge zwischen unterschiedlichen (Architektur-)Sichten hergestellt werden können. Zum zweiten veranschaulichen wir, in welchem Verhältnis die beiden topologischen Klassen, *interface* und *real*, von Komponenten stehen. Zu beachten ist, das wir Bezüge zwischen verschiedenen Architektursichten nur deshalb angeben können, weil in Form der Spezifikationsnetzwerke diese Sichten im Systemmodell *explizit* enthalten sind.

Beispiel 6.10 (Realisierungsarchitektur)

Für die Definition der Realisierungsarchitektur gehen wir von folgenden (hypothetischen) Annahmen und Anforderungen aus:

- Für Vormerkungs- und Ausleihe-Vorgänge existiere *eine* (COTS-)Komponente, welche sowohl die Steuerung von Vormerkungs- als auch von Ausleihe-Vorgängen realisiert. Diese Komponente sei bei der Realisierung des Bibliothekssystems einzusetzen.
- Weiterhin sei aus Gründen der Änderbarkeit die Schicht der Benutzerschnittstelle, in der Schnittstellenarchitektur aus Beispiel 6.8 durch die Komponente *BibSSI* repräsentiert, durch zwei separate Komponenten zu realisieren. Auf diese Weise bleiben etwa Änderungen rollenspezifischer Rechte lokal, wodurch die verbesserte Änderbarkeit erreicht wird.
- Alle anderen Realisierungskomponenten seien identisch zu den interface-Komponenten der Schnittstellenarchitektur aus Beispiel 6.8.

Damit ergibt sich die in Abbildung 6.9 dargestellte (Struktur der) Realisierungsarchitektur.

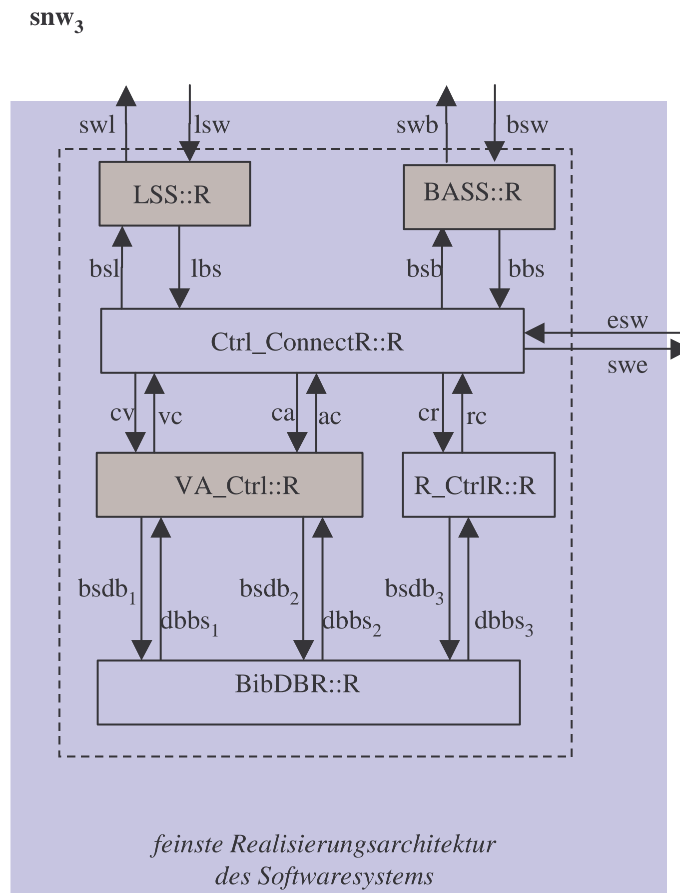


Abbildung 6.9: (Struktur der) Realisierungsarchitektur des Bibliothekssoftwaresystems.

Die Bezugnahmen auf die Schnittstellenarchitektur aus Beispiel 6.8 in den (drei) obigen, informellen Anforderungen spiegelt sich auch in der Form der folgenden

Spezifikation der Realisierungsarchitektur wider, wo wir die drei Anforderungen durch die drei Produkte $(p_{BibSS})_{pred}, (p_{VA})_{pred}$ und $(p_{id})_{pred}$[30] erfassen:

$$(SSARCH \cup RARCH)_{dnw}$$

wobei SSARCH die in Beispiel 6.8 definierte Produktmenge sei, und

$$RARCH =_{def}$$
$$\{(3, \{LSS, BASS, Ctrl_ConnectR, VA_Ctrl, R_CtrlR, BibDBR\})_{K_dnw},$$
$$(LSS, real)_{tt}, (BASS, real)_{tt},$$
$$(Ctrl_Connect, real)_{tt},$$
$$(VA_Ctrl, real)_{tt}, (R_Ctrl, real)_{tt},$$
$$(BibDBR, real)_{tt},$$
$$(LSS, I_{LSS}, O_{LSS})_{SS}, (LSS, I_{BASS}, O_{BASS})_{SS},$$
$$(V_Ctrl, I_{V_Ctrl}, O_{V_Ctrl})_{SS}, (A_Ctrl, I_{A_Ctrl}, O_{A_Ctrl})_{SS},$$
$$(p_{BibSS})_{pred},$$
$$(p_{VA})_{pred},$$
$$(p_{id})_{pred},$$
$$CHTYPE''\}$$

mit

$$p_{BibSS}, p_{VA}, p_{id} : SYS \to \mathbb{B}$$

wobei für alle $sys \in SYS$ gelte:

$$p_{BibSS}(sys) \Leftrightarrow_{def}$$
$$\{BibSSI, LSS, BASS\} \subseteq K_{sys} \wedge$$
$$\forall x \in TNW(CH).$$
$$(x = \{comp_{sys}(LSS), comp_{sys}(BASS)\}) \Rightarrow$$
$$(in(x), out(x), behav(x)) = component(comp_{sys}(BibSSI))$$

$$p_{VA_Ctrl}(sys) \Leftrightarrow_{def}$$
$$\{V_Ctrl, A_Ctrl, VA_Ctrl\} \subseteq K_{sys} \wedge$$
$$\forall x \in TNW(CH).$$
$$(x = \{comp_{sys}(V_Ctrl), comp_{sys}(A_Ctrl)\}) \Rightarrow$$
$$(in(x), out(x), behav(x)) = component(comp_{sys}(VA_Ctrl))$$

---

[30] bezüglich der Produktart *pred*, welche die für Spezifikationen in Form beliebiger Prädikate über der Menge SYS der Entwicklungssysteme, und damit für die allgemeinste Form von Spezifikation, steht, siehe Definition 2.5.

$$
\begin{aligned}
&p_{id}(sys) \Leftrightarrow_{def} \\
&\{Ctrl_Connect, Ctrl_ConnectR, R_Ctrl, R_CtrlR, BibDBI, BibDBR\} \subseteq K_{sys} \wedge \\
&(component(comp_{sys}(Ctrl_Connect)) \\
&= component(comp_{sys}(Ctrl_ConnectR))) \wedge \\
&(component(comp_{sys}(BibDBI)) = component(comp_{sys}(BibDBR))) \wedge \\
&(component(comp_{sys}(R_Ctrl)) = component(comp_{sys}(R_CtrlR)))
\end{aligned}
$$

SSARCH stellt eine Architektur gemäss Definition 6.8 dar. Anhand der durch SSARCH für das Netzwerk dnw_2 getroffenen Festlegung, bestimmen wir mit RARCH die Eigenschaften von dnw_3 so, dass SSARCH zusammen mit RARCH eine Softwarearchitektur (von dnw_3) gemäss Definition 6.8 ist. Um die Eigenschaft einer Architektur zu erfüllen, müssen wir die syntaktischen Schnittstellen der Realisierungskomponenten, die nicht identisch zu denen der Schnittstellenarchitektur sind, explizit definieren.

RARCH ist ein Beispiel dafür, wie wir Bezüge zwischen spezifikatorischen Sichten in unserem Ansatz herstellen können, um auf diese Weise eine Sicht aufbauend auf einer anderen Sicht, in diesem Fall die Realisierungsarchitektur aufbauend auf der Schnittstellenarchitektur, formulieren zu können. In der Produktart der *Software-Architektur* des V-Modells [BDI97] etwa, werden diese Bezüge unter dem Abschnitt *Anforderungszuordnung* erfasst. Dadurch wird der Zusammenhang zwischen Spezifikationen unterschiedlicher Hierarchieebenen der sogenannten *Erzeugnisstruktur*, dem im V-Modell vorgegebenen, hierarchisch strukturierten Modell des zu entwickelnden Systems, hergestellt.

□

7 Ausblick

An dieser Stelle kommen wir zu Fragestellungen, die im Umfeld dieser Arbeit liegen, aber in ihr nicht oder nur ansatzweise behandelt wurden.

Qualitätsanforderungen.

Mit der in dieser Arbeit gegebenen Form der Anforderungsspezifikation gehen wir nur auf Anforderungen ein, die üblicherweise unter dem Begriff der funktionalen Anforderungen zusammengefasst werden. Wesentlicher Bestandteil einer Anforderungsspezifikation ist aber auch die Angabe von Qualitätsanforderungen, wie beispielsweise Reaktionszeiten, Erweiterbarkeit und ähnlichen. Qualitätsanforderungen können beispielsweise eine wichtige Rolle bei der Auswahl von Entwurfsalternativen spielen. In existierenden Vorgehensmodellen werden Qualitätsanforderungen meist ohne klaren Bezug zu funktionalen Anforderungen formuliert. Durch eine Erweiterung unseres System- und Produktmodells um (die Spezifikation von) Qualitätsanforderungen können klare Bezüge hergestellt, und somit eine geeignete Grundlage für die im Requirements-Engineering bedeutsamen Auswahl- und Entscheidungsprozesse von Anforderungen geschaffen werden. In [DSV99] wird ein entsprechendes, entscheidungsorientiertes Produktmodell bereits informell skizziert.

Implementierung.

Neben den drei in dieser Arbeit behandelten, stellt auch die Softwareimplementierung eine primäre Entwicklungsaufgabe dar (vergleiche Abbildung 1.2). Da die Implementierung abhängig von spezifischen Technologien erfolgt, erscheinen technologiespezifische Erweiterungen von System- und Produktmodell notwendig. Im Zusammenhang mit (verteilten) Informationssystemen halten wir die Erweiterung um eine, an der relationalen Algebra orientierte Zustandssicht von Komponenten für sinnvoll, um auf diese Weise die Aufgabe des Datenbankentwurfs in den Gesamtansatz zu integrieren. Ein klarer Bezug zum Einsatz komponentenbasierter Implementierungstechnologien, beispielsweise COM- und CORBA-Komponenten (vergleiche [Box98] beziehungsweise [Gro98]), kann durch technologiespezifische Komponenten- und Spezifikationsarten erreicht werden.

Entscheidungsunterstützung.

Eine Entwicklung ist durch die Entscheidungen gekennzeichnet, die getroffen werden. Aus dem Bereich des Requirements-Engineerings sind entscheidungsunterstützende Prozesselemente bekannt (siehe etwa [RG94] und [Poh96]). In [DSV99] und [Dei01] werden Elemente zur Dokumentation von Entscheidungen, beispielsweise hierarchische Strukturen zur Kennzeichnung von Alternativen sowie zur Festlegung von Auswahlkriterien, in die Anforderungsspezifikation und den Architekturentwurf integriert. Auf ähnliche Weise könnte die Entscheidungsfindung im Rahmen des in dieser Arbeit vorgestellten Ansatzes unterstützt werden.

Sekundäre Entwicklungsaufgaben.

Für die Durchführung von Entwicklungsprojekten spielen auch sekundäre Aufgaben, wie das Projekt- und Konfigurationsmanagement eine wichtige Rolle. Durch geeignete Ergänzungen kann das in dieser Arbeit vorgestellte System- und Produktmodell zu einem Ansatz für eine umfassende Projektunterstützung erweitert werden. In [DSV99] werden beispielsweise Anforderungsspezifikation und Architekturentwurf mit der Aufgabe des Projektmanagements, auf informelle Weise, in Bezug gesetzt. [Dei01] gibt ein Produktmodell für das Versions- und Konfigurationsmanagement an, dass anhand des in dieser Arbeit gegebenen System- und Produktmodells zu einer spezifischen Variante konkretisiert werden kann, unter anderem durch Konkretisierung des Begriffs der Anforderung.

Methoden, Prozesse & Notationen.

Durch unser Produktmodell unterstützen wir eine systematische Entwicklung, indem wir mit Produktarten geeignete Schemata für die Charakterisierung von Entwicklungsinhalten angeben. Weitergehende Entwicklungsunterstützung kann in Form von Entwicklungsschritten und -prozessen gegeben werden, welche Anleitungen für das schrittweise Vorgehen bei der Erstellung von Produkten beschreiben. Aufgrund unseres konkreten und präzisen System- und Produktmodells wird es möglich, Vorgehensweisen sehr konkret anzugeben. Ein Beispiel wäre folgende Systematik für die Erstellung einer Spezifikation eines abstrakten Datentypen: Begonnen wird mit der Definition aller Selektorfunktionen, gefolgt von der Definition aller Konstruktorfunktionen, um anschließend für alle Konstruktorfunktionen deren Effekt bezüglich der Selektorfunktionen festzulegen oder entsprechende Abschichtungsregeln (auf andere Konstruktorfunktionen) anzugeben.

Die Anwendung bekannter, mathematischer Modelle der Informatik für unser Produkt- und Systemmodell erlaubt es, auch bei der Formulierung von Entwicklungsprozessen auf die zu diesen Modellen existierenden Methoden und Entwicklungstechniken zurückzugreifen. Hierbei sind gegebenenfalls entsprechende Anpassungen vorzunehmen. Beispielsweise wird in [Web91] eine Vorgehensweise für den Übergang von globalen Spurspezifikationen zu komponentenlokalen, funktionalen Verhaltensspezifikation vorgestellt. Diese Vorgehensweise kann auf unser spezielles Spurmodell, mit Interaktions- und Zustandsaspekt übertragen werden. Ähnliche Anpassungen sind notwendig, um etwa die in [BP00] gegebene Technik zur Verifikation von Lebendigkeits- und Sicherheitseigenschaften bezüglich Au-

tomatenspezifikationen auf unser Systemmodell und unsere Formen von Verhaltenseigenschaften, insbesondere auf Geschäftsprozessspezifikationen (vergleiche gegebenenfalls Abschnitt 4.1) zu übertragen

Weiterhin kann das Produktmodell als Grundlage für den systematischen Einsatz spezifischer Notationen dienen, indem die Semantik von Notationen anhand der Elemente der entsprechenden Produktarten formuliert wird. Wir haben dies bereits in dieser Arbeit anhand einiger Beispiele skizziert. In [Schw00] wird die grundsätzliche Korrelation von Produktarten und Notationen behandelt.

8 Grundlegende Definitionen

Dieses Kapitel enthält Definitionen von Begriffen und Konzepten, auf die wir uns in den Kapiteln 2 bis 6 beziehen, und die wir größtenteils aus anderen Arbeiten übernehmen.

8.1 Allgemeine Grundlagen

8.1.1 Ausgezeichnete Mengen

Mit $\mathbb{N}$ bezeichnen wir die Menge der natürlichen Zahlen.

$\mathbb{N} =_{\text{def}} \{ 0, 1, 2, 3, ..., n, n+1, ... \}$

$\mathbb{N}_+ =_{\text{def}} \mathbb{N} \setminus \{0\}$

Wir verwenden zudem die Menge der natürlichen Zahlen, erweitert um das Element ∞, durch das wir eine Zahl darstellen, die größer ist als jede natürliche Zahl:

$\mathbb{N}_\infty =_{\text{def}} \mathbb{N} \cup \{\infty\}$

Für ein geschlossenes Intervall zwischen einer Zahl $m \in \mathbb{N}_\infty$ und einer Zahl $n \in \mathbb{N}_\infty$ schreiben wir $[m, n]$; falls $m > n$, dann gelte $[m, n] =_{\text{def}} \varnothing$.

Die Menge der booleschen Werte stellen wir dar durch

$\mathbb{B} =_{\text{def}} \{ \text{true}, \text{false} \}$

true und *false* stehen für die booleschen Konstanten.

8.1.2 Relationen

Tupelschreibweise

Mit

$(e_1, e2, ..., en)$

bezeichnen wir das n-Tupel, welches e1 als erstes Element hat, e2 als zweites und so weiter. Für beliebige Mengen A1, ... , An steht

$A1 \times ... \times An$

für das kartesische Produkt der Mengen, das heißt für die Menge aller n-Tupel, deren j-tes Element aus Aj ist.

Wir führen weiterhin einen Projektionsoperator ein. Für jedes n-Tupel t und $1 \leq j \leq n$, bezeichnen wir mit $\Pi_j(t)$ das j-te Element von t.

Relation, Bild und Urbild

Seien S und T Mengen. Eine Teilmenge von $S \times T$ wird als (zweistellige) Relation bezeichnet. Für eine Menge $A \subseteq S$ ist das Bild von A (bezüglich einer Relation $R \subseteq S \times T$) definiert durch

$$R(A) =_{def} \{t \in T | \exists s \in A.(s,t) \in R\}.$$

Das Urbild einer Menge $B \subseteq T$ (bezüglich einer Relation $R \subseteq S \times T$) ist definiert durch

$$R^{-1}(B) =_{def} \{s \in S | \exists t \in B.(s,t) \in R\}.$$

Falls A beziehungsweise B einelementige Mengen mit A = {a} und B = {b} sind, schreiben wir abkürzend R(a) beziehungsweise $R^{-1}(b)$ für R({a}) beziehungsweise $R^{-1}(\{b\})$.

Partielle Ordnung

Für eine gegebene Menge P ist eine (reflexive) *partielle Ordnung* $\leq$ eine (zweistellige) Relation, also eine Teilmenge von $P \times P$, die reflexiv, antisymmetrisch, und transitiv ist. Eine partielle Ordnung ist *total*, falls für beliebige Elemente $a, b \in P$ entweder $a \leq b$ oder $b \leq a$ gilt.

8.1.3 Schreibweisen für Funktionen

$f : X \rightharpoonup Y$ bezeichnet die partielle Funktion f von X nach Y.

$f : X \to Y$ bezeichnet die totale Funktion f von X nach Y.

Die Menge aller partiellen beziehungsweise totalen Funktionen von X nach Y bezeichnen wir mit $(X \rightharpoonup Y)$ beziehungsweise $(X \to Y)$.

Für jede totale Funktion $f : X \rightarrow Y$ bezeichnen wir durch $f[s/t]$ die Funktion, die sich von f nur bei $s \in X$ unterscheidet und dort den Wert t liefert:

$$f[s/t](r) =_{def} \begin{cases} t & \text{falls } r = s, \\ f(r) & \text{sonst} \end{cases}$$

Für jede totale Funktion $f : X \rightarrow Y$ und jede Menge X' $\subseteq$ X bezeichnen wir durch $f|_{X'}$ die Einschränkung von f auf Argumente aus X':

$f|_{X'} : X' \rightarrow Y$

und für alle x $\in$ X'

$f|_{X'}(x) =_{def} f(x)$.

8.1.4 Prädikatenlogik

Wir verwenden die übliche Syntax und Semantik der Prädikatenlogik (vergleiche hierzu beispielsweise [LS84]).

8.1.5 Sequenzen und Ströme

Zu einer gegebenen Menge M von Elementen bezeichnen wir mit M^ω die Menge der Ströme über M. M^ω ist die Vereinigung der Menge der endlichen und unendlichen Sequenzen über M: $M^\omega =_{def} M^* \cup M^\infty$.

Für Ströme gibt es eine Reihe von hilfreichen Relationen und Operationen (seien s, t, u Ströme und a, b, c Elemente):

- ε bezeichnet den leeren Strom.
- $\langle s_1, ..., s_n \rangle$ bezeichnet den Strom, der die Elemente $s_1, ..., s_n$, in der angegebenen Reihenfolge, enthält.
- ft(s) ist nur definiert, falls $s \neq \varepsilon$ und liefert in diesem Fall das erste Element von s.
- rt(s) liefert den Strom, der sich durch Löschen des ersten Elements von s ergibt. Falls $s = \varepsilon$, dann gelte $rt(s) =_{def} \varepsilon$.
- a&s bezeichnet den Strom, dessen erstes Element a ist, gefolgt von s, so dass gilt ft(a&s) = a, und rt(a&s) = s.
- $s \circ t$ bezeichnet die Konkatenation von s und t. Falls s unendlich ist, dann liefert $s \circ t$ lediglich s. Oft schreiben wir $a \circ s$ für a&s und $a \circ b$ für $a \& b \& \varepsilon$, wobei wir Elemente mit Strömen der Länge 1 gleichsetzen. Es gilt $s \circ \varepsilon = \varepsilon \circ s = s$.

- $s \sqsubseteq t$ steht dafür, dass s *Präfix* von t ist. $\sqsubseteq$ ist ein Prädikat, angewandt in Infix-Schreibweise, dass genau dann gültig ist, falls $\exists u.\ s \circ u = t$. Das Prädikat definiert eine partielle Ordnung auf Strömen, die wir als Präfixordnung bezeichnen. Durch punktweise Anwendung erweitern wir die Präfixordnung auf Tupel von Strömen und auf Funktionen, die Ströme als Ergebnis liefern.
- *in(a, s)* steht für die Anwendung des Prädikats *in*, dass genau dann wahr ist, wenn *a* in *s* vorkommt.
- #s liefert die Länge von s, die auch ∞ sein kann („unendlich“).
- Für alle $n \in \mathbb{N}_\infty$ und s mit $\#s \geq n$, bezeichnen wir mit $s\uparrow n$ den Strom, den wir erhalten, wenn wir von s die ersten n Elemente entfernen. $s\uparrow\infty$ liefert den leeren Strom. $s\downarrow n$ definieren wir als das Präfix der Länge n von s. Für $n \in \mathbb{N}$ und $n \leq \#s$, bezeichnen wir mit s@n das n-te Element von s.
- a©s stellt die Filteroperation dar, die den Teilstrom von s liefert, der nur aus a Elementen besteht, beispielsweise gilt $a©\langle a,b,a,c\rangle = \langle a,a\rangle$. Als Verallgemeinerung kann der Operand auch eine Menge von Elementen sein, beispielsweise $\{a,b\}©\langle a,b,a,c\rangle = \langle a,b,a\rangle$.
- $s \prec t$ drückt aus, dass s ein Teilstrom von t ist, was formaler ausgedrückt bedeutet, dass gilt:

  $s = \langle s_0, s_1, \ \dots\ , s_n, \ \dots\ \rangle \ \Rightarrow$
  $\exists \quad \alpha_i, i \in \text{ Nat. } \alpha_0 \circ s_0 \circ \alpha_1 \circ s_1 \ \dots \ \circ\alpha_n \circ s_n \ \dots \ = t$

  Damit gilt beispielsweise $s \sqsubseteq t \Rightarrow s \prec t$, d.h. jedes Präfix ist auch Teilablauf.

Für eine formale Definition von Strömen und ihren Operationen verweisen wir auf [BS00].

8.2 Signaturen und Algebren

Definition 8.1 (Signatur, Teilsignatur)

Eine Signatur $\Sigma = (S, F)$ besteht aus

- einer Menge S (von Sorten),
- einer Menge F von (Bezeichnungen für) Funktionen, und
- einer Abbildung type : $F \to S^* \times S$.

Wir schreiben sorts(Σ) um S zu bezeichnen, opns(Σ) um F zu bezeichnen, und $f : w \rightarrow s$ um auszudrücken, dass für $f \in F$ gilt: $type(f) = (w,s)$. Die Abbildung type einer Signatur Σ bezeichnen wir mit $type_\Sigma$.

Eine Signatur $\Sigma = (S,F)$ mit $type_\Sigma : F \rightarrow S^* \times S$ bezeichnen wir als *Teilsignatur* einer Signatur $\Sigma' = (S',F')$ mit $type_{\Sigma'} : F' \rightarrow S'^* \times S'$, falls gilt, dass S ⊆ S' und F ⊆ F' sowie für alle f ∈ F gilt, dass $type_\Sigma(f) = type_{\Sigma'}(f)$.

Wir schreiben auch Σ ⊆ Σ', um auszudrücken, dass Σ Teilsignatur von Σ' ist.

□

Definition 8.2 (Term, Grundterm)

Jede Signatur $\Sigma = (S,F)$ legt eine Menge syntaktisch korrekter Ausdrücke fest, die aus freien Variablen und Funktionssymbolen der Signatur aufgebaut werden können.

Sei X eine S-indizierte Menge freier Variablen (die disjunkt zu den Konstanten in F ist). Für jede Sorte $s \in S$ ist die Menge $T(\Sigma,X)_s$ der *Terme der Sorte s* (die Elemente aus X enthält) die kleinste Menge, die folgende Elemente enthält:

- jedes $x \in X_s$ (der Sorte s) und jedes nullstellige Funktionssymbol $f \in F$ mit Typ → s, und
- jede Funktionsanwendung $f(t_1, \ldots, t_n)$, wobei $f : s_1, \ldots, s_n \rightarrow s$ aus F ist, und jedes t_i (i = 1, ... , n) ein Term (der Sorte s_i) aus $T(\Sigma,X)_{s_i}$ ist.

Terme ohne Elemente aus X nennen wir *Grundterme*, und für $T(\Sigma,\varnothing)_s$ schreiben wir auch $T(\Sigma)_s$.

□

Definition 8.3 (Σ-Algebra)

Sei Σ = (S,F) eine Signatur. Eine Σ-Algebra A besteht aus einer S-indizierten Menge von nichtleeren Trägermengen $\{A_s \mid s \in S\}$, und einer Funktion $f^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$ für jedes $f : s_1, \ldots, s_n \rightarrow s \in F$.

Falls alle $f^A \in F$ totale Funktionen sind, so sprechen wir von einer **totalen** Σ-Algebra.

Wir bezeichnen die Klasse aller Σ-Algebren mit *Alg(Σ)*.

□

Definition 8.4 (Signatur-Morphismus, σ-Redukt und Σ-Redukt)

Für zwei Signaturen $\Sigma = (S,F)$ und $\Sigma' = (S',F')$ ist ein Signatur-Morphismus $\sigma : \Sigma \to \Sigma'$ ein Paar $(\sigma_{sorts}, \sigma_{opns})$ wobei $\sigma_{sorts} : S \to S'$ und $\sigma_{opns} : F \to F'$ Abbildung derart sind, dass für alle $f : w \to s \in F$ gilt: $type(\sigma_{opns}(f)) = (\sigma^*(w), \sigma_{sorts}(s))$. Hier steht σ* für die Erweiterung von σ_{sorts} auf Zeichenketten über sorts; das heißt $\sigma^*(s_1, \ldots, s_n)$ steht für $\sigma_{sorts}(s_1), \ldots, \sigma_{sorts}(s_n)$ für $s_1, \ldots, s_n \in S$. Wir schreiben $\sigma(s)$ für $\sigma_{sorts}(s)$, $\sigma(w)$ für $\sigma^*(w)$, und $\sigma(f)$ für $\sigma_{opns}(f)$.

Zu einer gegebenen Σ'-Algebra A' und einem Signatur-Morphismus σ definieren wir das *σ-Redukt* von A', das wir auch mit $A'|_\sigma$ bezeichnen, als die Σ-Algebra mit den Trägermengen $(A'|_\sigma)_s =_{def} A'_{\sigma(s)}$ für alle $s \in sorts(\Sigma)$ und $f^{A'|_\sigma} =_{def} \sigma(f)^{A'}$ für alle $f \in opns(\Sigma)$.

Falls σ die Identität auf Σ ist, und damit $\Sigma \subseteq \Sigma'$, bezeichnen wir $A'|_\sigma$ auch mit $A'|_\Sigma$. Die Σ-Algebra $A'|_\Sigma$ nennen wir auch *Σ-Redukt* von A'.

Durch elementweise Reduktbildung erweitern wir den Reduktbegriff auf Mengen von Algebren. Für jede Menge C von Σ'-Algebren definieren wir $C|_\sigma =_{def} \{A'|_\sigma \mid A' \in C\} \subseteq Alg(\Sigma)$.

□

Definition 8.5 (Σ-Homomorphismus)

Seien A und B Σ-Algebren, mit $\Sigma = (S,F)$. Ein *Σ-Homomorphismus* $h : A \to B$ ist eine Familie von Abbildungen $\{h_s : A_s \to B_s \mid s \in S\}$ derart, dass für alle $f : s_1, \ldots, s_n \to s \in F$ und alle $a_1 \in A_{s_1}, \ldots, a_n \in A_{s_n}$ gilt:

$$h_s(f^A(a_1, \ldots, a_n)) = f^B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)).$$

Wir bezeichnen zwei Σ-Algebren A und B als *homomorph*, falls ein Homomorphismus $h : A \to B$ oder $h : B \to A$ existiert.

□

Definition 8.6 (Erreichbarkeit, Σ-Rechenstruktur, Abstrakter Datentyp)

Für eine Signatur $\Sigma = (S,F)$ und eine S-indizierten Menge X freier Variablen bildet die sogenannte *Termalgebra*, die wir auch mit $T(\Sigma,X)$ bezeichnen, eine Σ-Algebra mit Trägermenge $T(\Sigma,X)_s$ für jede Sorte $s \in S$, und $f^{T(\Sigma,X)}(t_1, \ldots, t_n) =_{def} f(t_1, \ldots, t_n)$ für jedes $f : s_1, \ldots, s_n \to s \in F$ und $t_i \in T(\Sigma,X)_{s_i}$ (i = 1, ... , n). Durch $T(\Sigma) =_{def} T(\Sigma,\varnothing)$ definieren wir die sogenannte *Grundterm-Algebra*.

Falls A eine Σ-Algebra und X eine sorts(Σ)-indizierte Menge ist, nennen wir eine Abbildung $v : X \rightarrow A$ eine *Belegung*, falls gilt $v(x) \in A_s$ für alle $x \in X_s$, $s \in \text{sorts}(\Sigma)$.

Zwischen einer Signatur $\Sigma = (S,F)$ und einer S-indizierten Menge X freier Variablen können wir über den Begriff der *Interpretation* einen Bezug zu einer Σ-Algebra A herstellen. Sei $v : X \rightarrow A$ eine Belegung. Die Interpretation eines Terms t in A (bezüglich v) ist gegeben durch $v^* : T(\Sigma,X) \rightarrow A$, die wir wie folgt definieren:

i. $v^*(x) =_{\text{def}} v(x)$ für jedes $x \in X$,

ii. $v^*(f(t_1, \ldots, t_n)) =_{\text{def}} f^A(v^*(t_1), \ldots, v^*(t_n))$ für jedes $f : s_1, \ldots, s_n \rightarrow s \in F$, und $t_i \in T(\Sigma,X)_{s_i}$ (i = 1, ... , n).

Falls ein Term t ein Grundterm ist, ist die Interpretation unabhängig von v und eindeutig definiert; wir schreiben auch t^A für $v^*(t)$.

Sei $A \in \text{Alg}(\Sigma)$, $S \subseteq \text{sorts}(\Sigma)$, $F \subseteq \text{opns}(\Sigma)$. Wir nennen die Algebra *A erreichbar auf S mit F*, genau dann wenn gilt, dass es für alle $s \in S$ und $a \in A_s$ einen Term $t \in T\left(\Sigma',(X_s)_{s \in \text{sorts}(\Sigma)}\right)$ und eine Belegung v gibt, derart dass $v^*(t) = a$, wobei $\Sigma' = (\text{sorts}(\Sigma),F)$.

Wenn A erreichbar auf sorts(Σ) mit opns(Σ) ist, dann nennen wir A *termerzeugt*. Eine termerzeugte Σ-Algebra nennen wir auch *Σ-Rechenstruktur* (oder Σ-Datentyp). Die Klasse aller Σ-Rechenstrukturen bezeichnen wir mit *Gen(Σ)*.

Als *abstrakten Datentyp* bezeichnen wir eine Isomorphieklasse einer Σ-Rechenstruktur.

□

Im Unterschied zu Definitionen der Termerzeugtheit, die auf der Interpretation von Grundtermen basieren, erlaubt uns das oben eingeführte allgemeine Konzept der Erreichbarkeit Algebren zu charakterisieren, die sowohl termerzeugte als auch nicht termerzeugte Trägermengen umfassen. Zudem können wir so einzelne Sorten und deren Konstruktoren auszeichnen (vergleiche [Bre91]). Damit steht uns ein modularer Ansatz für Aspekte der Termerzeugtheit zur Verfügung.

Definition 8.7 (Boolesche Algebra)

Wir definieren die Signatur und Axiome der Klasse der booleschen Algebren (Algebren der Wahrheitswerte).

Die boolesche Signatur Σ_{BOOL} definieren wir durch

sorts(Σ_{BOOL}) $=_{def}$ *{Bool}*

opns (Σ_{BOOL}) $=_{def}$ *{true, false, not, or, and}*

wobei gelte

true : $\rightarrow$ *Bool*

false : $\rightarrow$ *Bool*

not : *Bool* $\rightarrow$ *Bool*

or : *Bool, Bool* $\rightarrow$ *Bool*

and : *Bool, Bool* $\rightarrow$ *Bool*

Die Klasse der booleschen Algebren C_{BOOL} definieren wir als Teilmenge der Σ_{BOOL}-Rechenstrukturen, das heißt $C_{BOOL} \subseteq Gen(\Sigma_{BOOL})$. Dabei gelte für jedes Element aus C_{BOOL}, dass die Trägermenge der Sorte Bool durch true und false erreichbar ist und, dass für alle x ∈ Bool folgende Axiome erfüllt sind:

true ≠false

not(true) = false

not(not(x)) = x

or(false, x) = or(x, false) = x

or(true, x) = or(x, true) = true

and(true, x) = and(x, true) = x

and(false, x) = and(x, false) = false

□

Wir führen den Begriff der *Entitätssignatur* ein. Eine Entitätssignatur betrachten wir immer bezüglich einer gegebenen Menge von Sorten.

Definition 8.8 (Entitätssignatur)

Eine Signatur Σ mit sorts(Σ) = S ist eine *Entitätssignatur* bezüglich einer Menge (von Sorten) V, falls gilt:

i. $S = V \cup ID$, wobei $ID =_{def} \{id_s \mid s \in V\}$

ii. $V \cap ID = \emptyset$

Mit obigen Axiomen legen wir fest, dass eine Entitätssignatur bezüglich einer Menge V von Sorten, zu jeder Sorte s ∈ V eine (Identifikator-)Sorte id_s umfasst. Die Menge der Identifikator-Sorten beschreiben wir durch die V-indizierte Menge ID.

□

8.3 Prozesse

Kausalrelationsmodelle erlauben die Beschreibung von Systemverhalten mit „echter Nebenläufigkeit“. Wir führen Prozesse als eine Form eines Kausalrelationsmodells ein. Prozesse entsprechen den *action structures* aus [Bro89]. Damit unterscheiden sich Prozesse von den in [Pra86] definierten *labelled partial orders* (lpo) nur darin, dass das Alphabet der Ereignismarkierungen nicht (expliziter) Bestandteil eines Prozesses ist.

In [GG98] wird die Semantik der verschiedenen Arten von Ereignis-Strukturen (event structures) auf Basis von Mengen von *configuration structures* angegeben. Configuration structures haben große Ähnlichkeit mit lpo's. Das macht deutlich, dass lpo's und somit unser Prozessmodell eine geeignete Abstraktion für Ereignis-Strukturen sind:

> *„Although the refinement operators we have defined depend on the particular “syntax” of the chosen event structures, in all cases the relevant behaviour of an event structure is determined by its set of configurations […].”* [GG98]

Anschließend an die Definition des Prozess-Begriffes beschreiben wir den Zusammenhang zwischen Prozessen und einem strombasierten Modell. Grundsätzlich gibt es unterschiedliche Arten, um von einem Kausalrelationsmodell zu einem strombasierten Modell zu kommen. Als Zwischenschritt dieses Übergangs interpretieren wir einen Prozess als Transitionssystem. Die Ausführungsfolgen des Transitionssystems verstehen wir als die Realisierung des Prozesses im strombasierten Modell. Abhängig von der Wahl des Transitionssystems ordnen wir Prozessen unterschiedliche Ausführungsfolgen zu. In [GG98] wird zwischen einer reinen Interleaving- und einer Step-Semantik unterschieden. In ersterer entspricht eine Transition der „Ausführung“ eines (Prozess-)Ereignisses. Im Falle der Step-Semantik, der Ansatz auf den wir im Folgenden eingehen, entspricht eine Transition der parallelen Ausführung mehrerer (kausal unabhängiger) Ereignisse.

Definition 8.9 (Prozess)

Gegeben sei eine Menge E von Ereignissen und eine Menge M von (Ereignis-)Markierungen. Ein Tripel

$$p = (E_p, \leq_p, \alpha_p)$$

nennen wir einen *Prozess (über E und M)*, falls folgende Aussagen gelten:

- $E_p \subseteq E$,
- $\leq_p$ ist eine partielle Ordnung über E_p, und
- α_p ist eine Markierungsabbildung der Form $E_p \rightarrow M$.

Wir sprechen von Prozessen mit einer *diskreten Kausalitätsordnungen*, wenn wir zu jedem Ereignis dessen unmittelbare Nachfolger angeben können. Wir nennen ein Ereignis e_1 *unmittelbaren Nachfolger* von e_0, falls für alle $e \in E_p$ gilt: $e_0 \leq_p e \leq_p e_1 \Rightarrow (e = e_1 \vee e = e_0)$. Zwischen einem Ereignis und einem unmittelbaren Nachfolger liegen also keine weiteren Ereignisse.

Falls für jedes Ereignis $e \in E_p$ die Menge der kausalen Vorgänger $\{d \in E_p : d \leq_p e\}$ endlich ist, sprechen wir von einem *endlich fundierten Prozess*. Damit ist die dem Prozess zugrunde liegende Ordnung noethersch, da keine unendlich absteigenden Ketten von Ereignissen existieren.

Wir bezeichnen die Menge aller Prozesse über einer Menge von Ereignissen E und einer Menge von Markierungen M mit *pcs(E,M)*. Die Menge der endlichen fundierten Prozesse bezeichnen wir mit $pcs_{fin}(E,M)$. Die Menge der endlichen Prozesse bezeichnen wir mit $pcs^*(E,M)$. Den *leeren Prozess*, dessen Ereignismenge die leere Menge ist, bezeichnen wir mit $p_\varnothing$.

□

Definition 8.10 (Teilprozess und Präfix, endliche Approximation, Prozess-Subtraktion)

Seien $p_1 = (E_1, \leq_1, \alpha_1)$ und $p_2 = (E_2, \leq_2, \alpha_2)$ Prozesse. Gilt folgende Beziehung für p_1 und p_2:

$$E_1 \subseteq E_2,$$

$$\alpha_{2|E_1} = \alpha_1,$$

$$\leq_{2|E_1 \times E_1} = \leq_1,$$

so heißt p_1 *Teilprozess* von p_2 und wir schreiben $p_1 \prec p_2$. Gilt zusätzlich

$$\forall e \in E_2, d \in E_1 : e \leq_2 d \Rightarrow e \in E_1$$

so heißt der Prozess p_1 *Präfix* von p_2. Wir schreiben dann

$$p_1 \sqsubseteq p_2.$$

Ein Präfix enthält einen Teilablauf *vollständig* bezüglich der Kausalordnung (Links-Abgeschlossenheit).

Um Teilprozesse eines gegebenen Prozesses bilden zu können, definieren wir die sogenannte *Prozess-Subtraktion*

$$. \setminus . : pcs_{fin}(E,M) \times \wp(E) \rightarrow pcs_{fin}(E,M)$$

wobei E eine Ereignismenge und M eine Menge von Ereignismarkierungen sei. Für jeden Prozess $p = (E_p, \leq_p, \alpha_p) \in pcs_{fin}(E,M)$ und $X \in \wp(E)$ gelte

$$p \setminus X =_{def} \left(E_{px}, \leq_p |_{E_{px} \times E_{px}}, \alpha_p |_{E_{px}} \right)$$

wobei $E_{px} = E_p \setminus X$ sei. Für einen gegebenen Prozess q schreiben wir auch $p \setminus q$ für

$$p \setminus \{e | e \in \Pi_1(q)\}.$$

□

Definition 8.11 (Operationen auf Prozessen)

Analog zu Operationen auf Strömen, definieren wir einige Operationen auf Prozessen. Sei E eine Menge von Ereignisbezeichnern und M eine Menge von Ereignismarkierungen. Sei $m \in M$ und $p \in pcs_{fin}(E,M)$.

- *Projektion*

  $$©: M \times pcs_{fin}(E,M) \rightarrow pcs_{fin}(E,M)$$

  mit

  $$©(m,p) =_{def} p \setminus \{e \in \Pi_1(p) | \Pi_3(p)(e) \neq m\}$$

- *Mächtigkeit*

  $$\#: pcs_{fin}(E,M) \rightarrow \mathbb{N}^{\infty}$$

  mit

  $$\#(p) =_{def} |\Pi_1(p)|$$

□

Definition 8.12 (Minimale Ereignisse)

Für jede Ereignismenge E und Markierungsmenge M sowie jeden Prozess $p = (E_p, \leq_p, \alpha_p) \in pcs_{fin}(E,M)$ bezeichnen wir mit

$$min(p) : pcs_{fin}(E,M) \rightarrow \wp(E)$$

die Menge der *minimalen Ereignisse* des Prozesses, die wir definieren durch

$$min((E_p, \leq_p, \alpha_p)) =_{def} \{e \in E_p | \forall e' \in E_p \setminus \{e\}. \neg(e' \leq_p e)\}.$$

□

Definition 8.13 (Isomorphe Prozesse)

Seien $p = (E_p, \leq_p, \alpha_p)$ und $q = (E_q, \leq_q, \alpha_q)$ zwei Prozesse über E und M. p und q sind *isomorph*, genau dann wenn es eine Bijektion f zwischen E_p und E_q ($f: E_p \rightarrow E_q$) gibt, welche die Markierungsabbildung sowie die Kausalrelation erhält, das heißt falls

$$\forall e \in E_p.\ \alpha_p(e) = \alpha_q(f(e)), \text{ und}$$

$$\forall e, e' \in E_p.\ e \leq_p e' \Leftrightarrow f(e) \leq_q f(e')$$

gilt. Wir schreiben $p \cong q$.

Mit $[p]$ bezeichnen wir die durch einen Prozess p charakterisierte Klasse isomorpher Prozesse. Die Menge aller Isomorphieklassen von Prozessen über E und M bezeichnen wir mit $[pcs(E,M)]$.

Für eine Prozessmenge $P \in \wp(pcs_{fin}(E,M))$ bezeichnen wir mit *[P]* die Menge der Isomomorphieklassen, die wir durch elementweise Isomorphieklassenbildung bezüglich aller Elemente aus P erhalten.

□

Prozesse sind damit ähnlich zu den in [Pra86] vorgestellten *labelled partial orders* (lpos), Isomorphieklassen von Prozessen sind ähnlich zu *pomsets* (*partially ordered multisets*). Im Unterschied zu oben eingeführtem Prozessbegriff, umfassen lpos und pomsets in [Pra86] das Alphabet, welches den Bildbereich der Markierungsabbildung darstellt.

Definition 8.14 (Operationen auf Isomorphieklassen von Prozessen)

Seien eine Ereignismenge E und eine Menge M von Ereignismarkierungen gegeben. Für zwei Prozesse $(E_p, \leq_p, \alpha_p), (E_q, \leq_q, \alpha_q) \in pcs(E,M)$, für die wir ohne Einschränkung der Allgemeinheit (da wir nur Isomorphieklassen betrachten) annehmen, dass E_p und E_q disjunkt sind, definieren wir:

Parallelkomposition

$$\|: [pcs(E,M)] \rightarrow [pcs(E,M)]$$

durch

$$[E_p, \leq_p, \alpha_p] \,\|\, [E_q, \leq_q, \alpha_q] =_{def} [E_p \cup E_q, \leq_p \cup \leq_q, \alpha_p \cup \alpha_q].$$

Konkatenation / Sequentielle Komposition

$$; : [pcs(E,M)] \rightarrow [pcs(E,M)]$$

durch

$$[E_p, \leq_p, \alpha_p];[E_q, \leq_q, \alpha_q] =_{def} \begin{cases} [E_p \cup E_q, \leq_p \cup \leq_q \cup (E_p \times E_q), \alpha_p \cup \alpha_q], \\ \text{falls } E_p \text{ endlich,} \\ [E_p, \leq_p, \alpha_p], \text{ sonst} \end{cases}$$

Die beiden Kompositionsoperatoren erweitern wir, durch punktweise Anwendung auf Mengen von Isomorphieklassen, das heißt für alle $P, Q \in \wp([pcs(E,M)])$ definieren wir

$$P \,\|\, Q =_{def} \{[r], r \in pcs(E,M) |\; p \in P, q \in Q.\; [r] = p \,\|\, q\}$$

und

$$P;Q =_{def} \{[r], r \in pcs(E,M) |\; p \in P, q \in Q.\; [r] = p;q\}.$$

□

In [Win82] wird die Parallelkomposition von Ereignis-Strukturen, genauer von *prime event structures*, bezüglich einer Synchronisations-Algebra L definiert. Damit kann die Semantik für Sprachen definiert werden, in denen die Synchronisation über gemeinsame Ereignisse, genauer über Ereignisse die mit Elementen eines

ausgezeichneten Alphabets markiert sind, spezifiziert werden kann. Im Rahmen dieser Arbeit betrachten wir nur die unsynchronisierte Parallelkomposition, so dass der oben eingeführte Kompositionsoperator ausreicht.

Definition 8.15 (Transitionssystem)

Wir beschreiben ein *Transitionssystem* durch ein Tupel (Ac, C, $\rightarrow$, Init), mit

Ac Menge der Aktionen

C Menge der Konfigurationen

$\rightarrow \subseteq C \times Act \times C$, Transitionsrelation

Init $\subseteq C$, initiale Konfigurationen.

Einen Aktionsstrom $\langle a_1, a_2, a_3, ... \rangle \in Ac^\omega$ bezeichnen wir als *Aktionsfolge* des Transitionssystems, genau dann wenn es X_0, X_1, X_2 ... $\in$ C gibt, so dass gilt

$X_0 \in$ Init

$(X_i, a_{i+1}, X_{i+1}) \in \rightarrow$, für alle i.

Durch

$$X_0 \xrightarrow{a_1} X_1 \xrightarrow{a_2} X_2 ...$$

stellen wir die zugehörige *Ausführungsfolge* dar. Wir bezeichnen eine Ausführungsfolge φ als *vollständig*, falls gilt

φ ist unendlich, oder

$\varphi = X_0 \xrightarrow{a_1} X_1 \xrightarrow{a_2} X_2 ... X_{n-1} \xrightarrow{a_n} X_n$ und

$\neg(\exists X \in C, a \in Ac.(X_n, a, X) \in \rightarrow)$.

Mit *executions*(trans) (Aktionsfolgen) bezeichnen wir die Menge aller Aktionsfolgen eines Transitionssystems trans. *cexecutions*(trans) (vollständige Aktionsfolgen) steht für die Menge der *Aktionsfolgen der vollständigen Ausführungsfolgen* von trans.

□

Definition 8.16 (Step-Semantik von Prozessen)

Unter der Step-Semantik eines Prozesses p = (E_p, $\leq_p$, α_p), p $\in$ pcs_{fin}(E,M) verstehen wir das Transitionssystem

Step(p) $=_{def}$ *(Ac_p, C_p, $\rightarrow_p$, $Init_p$)*

mit

- $Ac_p =_{def} (M \rightarrow \mathbb{N})$
- $C_p =_{def} \{ c \in \wp(E_p) | (c, \leq_p|_c, \alpha_p|_c) \sqsubseteq p \}$

- $\rightarrow_p =_{def} \{(c,a,c') \in (C_p \times Ac \times C_p) \mid (c \subset c') \wedge (\forall e,d \in c' \setminus c . co(e,d)) \wedge (a = l_p(c' \setminus c))\}$
- $Init_p =_{def} \{\emptyset\}$

wobei wir hier mit

$co_p =_{def} \{(e,d) \in (E_p \times E_p) \mid \neg(e \leq_p d \wedge d \leq_p e)\}$

unabhängige Ereignisse auszeichnen. Cp steht für die Menge der Ereignismengen der Präfixe von p und

$l_p : \wp(E_p) \rightarrow (M \rightarrow \mathbb{N})$ mit

$l_p(G)(m) =_{def} |\{e \in G \mid \alpha_p(e) = m\}|$

die Multimenge der Markierungen der Ereignisse aus einer (Ereignis-)Menge G. Wir modellieren hier Multimengen über M durch Abbildungen der Form ($M \rightarrow \mathbb{N}$).

□

8.4 Komponenten, Netzwerke und Automaten

In der komponentenbasierten Perspektive des in Kapitel 2 vorgestellten Systemmodells beschreiben wir ein System als ein Netzwerk autonom agierender, miteinander interagierender Einheiten, die wir Komponenten nennen. Diese Sichtweise eines verteilten Systems halten wir für geeignet, um Informationssysteme und deren Rolle im Anwendungskontext zu erfassen.

In diesem Abschnitt führen wir die grundlegenden Konzepte der komponentenbasierten Perspektive des Systemmodells ein.

Als Grundlage verwenden wir das Modell, welches in der Methode FOCUS [BS00] eingeführt wird. FOCUS ist ein allgemeiner Rahmen zur Entwicklung verteilter Systeme und kann überall dort eingesetzt werden, wo Systeme aus räumlich oder konzeptuell verteilten Komponenten zusammengesetzt sind. Zur Spezifikation unterschiedlicher Aspekte eines Systems werden in FOCUS unterschiedliche Beschreibungstechniken angeboten. Die Semantik der Beschreibungstechniken wird anhand eines Modells gegeben, in dem die Struktur eines verteilten Systems durch Komponenten und deren Verknüpfung über gerichtete Kanäle festgelegt ist. Für jeden Kanal ist die Menge der Nachrichten festgelegt, die zulässigerweise übertragen werden. Es wird davon ausgegangen, dass nur diese Nachrichten übertragen werden. Auf jedem Kanal können Nachrichten nur unidirektional und rein sequentiell übertragen werden. In dieser Arbeit beschränken wir uns auf Punkt-zu-Punkt Verbindungen, das heißt, ein Kanal verbindet immer genau eine sendende mit einer empfangenden Komponente. Broadcasting, beispielsweise, muss mittels entsprechender Broadcast-Komponenten modelliert werden. Nachrichtenaustausch erfolgt auf nachrichten-asynchrone Weise, das heißt ein Sender kann Nachrichten immer

ungehindert senden und nicht durch den Empfänger blockiert werden. Dies entspricht der Annahme unbeschränkter Pufferkapazitäten.

Eine Komponente ist zum einen durch die Menge seiner Ein- und Ausgabekanäle, die unter dem Begriff der syntaktischen Schnittstelle zusammengefasst sind, charakterisiert. Da von einer statischen Systemstruktur ausgegangen wird, bleibt die syntaktische Schnittstelle einer Komponente über dessen Lebensdauer invariant. Das zweite Charakteristikum einer Komponente ist dessen Verhalten. Das beobachtbare Verhalten sind die Ausgaben, die eine Komponente in Abhängigkeit von erhaltenen Eingaben macht. Weiterhin wird davon ausgegangen, dass eine Komponente keinen Einfluss darauf hat, welche Eingaben gemacht werden (mit der Ausnahme rückgekoppelter Kanäle).

Im Folgenden führen wir als Grundlage der mathematischen Verhaltensbeschreibung von Komponenten zunächst das Konzept der *gezeiteten Ströme* ein. Durch gezeitete Ströme von Nachrichten modellieren wir die über einen Kanal ausgetauschten Nachrichten.

Darauf aufbauend definieren wir Komponenten, bestehend aus ihrer Schnittstelle und einer Verhaltensfunktion, die Eingabenachrichtenströme auf Ausgabeströme abbildet. Anschließend daran, definieren wir Netzwerke als endliche Menge von Komponenten.

Um *zustandsbezogenes* (Interaktions-)Verhalten von Komponenten und Netzwerken angeben zu können, führen wir, in Abschnitt 8.4.2, die Menge der sogenannten *Interaktions-Zustands-Tupel* und –Ströme mit einem speziellen Operator zur Parallelkomposition ein. Interaktions-Zustands-Tupel sind 2-Tupel, bestehend aus einem Zustand und einer Kanalbelegung. Interaktions-Zustands-Ströme verwenden wir unter anderem, um die Ausführungsfolgen von Automaten anzugeben. Zudem dienen sie der Formulierung der Aufgabenstellung, die eine Komponente oder ein Netzwerk zu lösen, das heißt zu generieren, hat.

Um das Verhalten einer Komponente auf operationelle Weise angeben zu können, sind Automaten ein geeignetes Mittel. Wir definieren eine zu unserem Komponentenbegriff passende Automatenart in Abschnitt 8.4.3.

8.4.1 Komponenten

Wie erwähnt, modellieren wir Interaktion zwischen Komponenten durch Nachrichtenaustausch (über Kanäle). Um Kausalität zwischen Interaktionen auf einfache und damit anschauliche Weise modellieren zu können, gehen wir in dieser Arbeit von einem (zeitlich) *getakteten Verhalten* aus. Wir nehmen einen systemweiten, diskreten (Zeit-)Takt an. Pro Takt werden Nachrichten empfangen und gesendet. Die Dauer eines Taktes ist beliebig, aber beschränkt, so dass pro Takt nur endlich viele Nachrichten ausgetauscht werden. Die auf einem Kanal pro Takt übertragenen Nachrichten erfassen wir daher in Form einer *endlichen* Sequenz von Nachrichten. Der Spezialfall der leeren Sequenz, beispielsweise, erfasst den Fall, dass in einem Takt keine Nachricht übertragen wurde. Da Zeit immer fortschreitet und wir (vereinfachend) von einer unbegrenzten Lebensdauer eines Systems ausgehen, modellieren wir die auf einem Kanal über die gesamte Systemlebensdauer übertragenen Nachrichten durch unendliche Sequenzen endlicher Nachrichtensequenzen.

Diese sogenannten Kanalhistorien enthalten damit Information darüber, in welchem Takt Nachrichten ausgetauscht wurden. Zum Beispiel bedeutet die Kanalhistorie

<<a>,ε,<bc>, ...>

dass im ersten Takt die Nachricht *a*, im zweiten Takt keine Nachricht und im dritten Takt die Nachrichtenfolge *<bc>* übertragen wurden.

Wir bezeichnen die Menge der endlichen und unendlichen Sequenzen über endlichen Sequenzen (über einer gegebenen Elementmenge) als die Menge der gezeiteten Ströme:

Definition 8.17 (gezeitete Ströme)

Sei M eine Menge von Elementen. Die Menge

$$(M^*)^\infty$$

bezeichnen wir als die Menge der *gezeiteten Ströme* über M.

□

Wie erwähnt, legen wir für jeden Kanal die Menge der Nachrichten fest, die zulässigerweise übertragen werden. Als *Kanalbelegung* bezeichnen wir eine, dieser Anforderung entsprechende, Zuordnung unendlicher, gezeiteter Ströme zu Kanalbezeichnern. So können wir beispielsweise durch eine Kanalbelegung der Eingabekänale einer Komponente die Eingaben erfassen, die eine Komponente über ihre Lebensdauer empfangen hat.

Definition 8.18 (Kanalbelegung und Interaktionsmuster)

Sei M eine Menge von Nachrichten und CH eine (endliche) Menge von Kanälen, die durch eine Abbildung ctype : CH → ℘(M) typisiert sind. Die Menge der Kanalbelegungen von CH definieren wir durch

$$\overrightarrow{CH} =_{def} \left\{ x \in CH \rightarrow (M^*)^\infty \,\middle|\, \forall c \in CH.x(c) \in (ctype(c)^*)^\infty \right\}$$

Eine Kanal-Belegung $x \in \overrightarrow{CH}$ ordnet jedem Kanal $c \in CH$ einen gezeiteten Strom von Elementen aus ctype(c) zu.

Die Menge der Interaktionsmuster von CH definieren wir durch

$$CH^{\vec{*}} =_{def} \{ x \in CH \rightarrow M^* | \forall c \in CH.x(c) \in (ctype(c)^*) \}$$

□

Eine Komponente ist strukturell charakterisiert durch die Menge seiner Ein- und Ausgabekanäle. Das beobachtbare Verhalten einer Komponente ist charakterisiert durch die Korrelation zwischen (empfangenen) Eingaben und (gesendeten) Ausgaben. Empfangene Eingaben modellieren wir durch Kanalbelegungen der Eingabekanäle, gesendete Ausgaben durch Kanalbelegungen der Ausgabekanäle. Wir sprechen im folgenden auch von Eingabe- und Ausgabehistorien. Das Verhalten einer

Komponente geben wir durch eine Funktion an, die jeder Eingabehistorie eine Menge von Ausgabehistorien zuordnet. Indem wir einer Eingabe- eine *Menge* von Ausgabehistorien zuordnen, können wir *nichtdeterministisches Komponentenverhalten* modellieren.

In der Realität können die Ausgaben einer Komponente nur durch die vorangegangenen Eingaben bestimmt werden, nicht aber durch der Ausgabe nachfolgende Eingaben. Um nur „reale" Verhalten modellieren zu können, fordern wir von der Verhaltensfunktion einer Komponente *strikte Kausalität*:

Definition 8.19 (Komponenten)

Sei CH eine (endliche) Menge typisierter Kanäle. Die Menge der (streng kausalen) Komponenten über CH definieren wir durch

$$\begin{aligned}\mathrm{KP}(\mathrm{CH}) =_{\mathrm{def}} \Big\{(\mathrm{I},\mathrm{O},\mathrm{F}) \in \wp(\mathrm{CH}) \times \wp(\mathrm{CH}) \times \Big(\overrightarrow{\wp(\mathrm{CH})} \to \wp\Big(\overrightarrow{\wp(\mathrm{CH})}\Big)\Big) \Big| \\ \mathrm{F} \in \Big(\vec{\mathrm{I}} \to \wp\Big(\vec{\mathrm{O}}\Big)\Big) \wedge \mathrm{causal}(\mathrm{F})\Big\}\end{aligned}$$

wobei gelte:

$$\begin{aligned}&\mathrm{causal}(\mathrm{F}) \Leftrightarrow_{\mathrm{def}} \\ &\forall \mathrm{i} \in \mathbb{N}, \mathrm{x}, \mathrm{z} \in \vec{\mathrm{I}}.\mathrm{x} \downarrow \mathrm{i} = \mathrm{z} \downarrow \mathrm{i} \Rightarrow \mathrm{F}(\mathrm{x}) \downarrow (\mathrm{i}+1) = \mathrm{F}(\mathrm{z}) \downarrow (\mathrm{i}+1)\end{aligned}$$

Die Elemente I und O einer Komponente (I, O, F) bezeichnen wir als die *syntaktische Schnittstelle* der Komponente, bestehend aus der Menge I der *Eingabe*- und der Menge O der *Ausgabekanäle*. Für alle Komponenten $\mathrm{c} \in \mathrm{KP}(\mathrm{CH})$ führen wir folgende Bezeichnungen ein:

$$\begin{aligned}&\mathrm{in}(\mathrm{c}) =_{\mathrm{def}} \Pi_1(\mathrm{c}), \\ &\mathrm{out}(\mathrm{c}) =_{\mathrm{def}} \Pi_2(\mathrm{c}), \\ &\mathrm{behav}(\mathrm{c}) =_{\mathrm{def}} \Pi_3(\mathrm{c})\end{aligned}$$

□

Das Prädikat *causal()* entspricht der in [**BS00**] als *strong causality* bezeichneten Eigenschaft.

Durch Komposition bilden wir eine Komponente aus gegebenen Komponenten, indem wir von internen Kanälen, das heißt Kanälen die zu komponierende Komponenten verbinden, und von Interaktion auf diesen internen Kanälen abstrahieren.

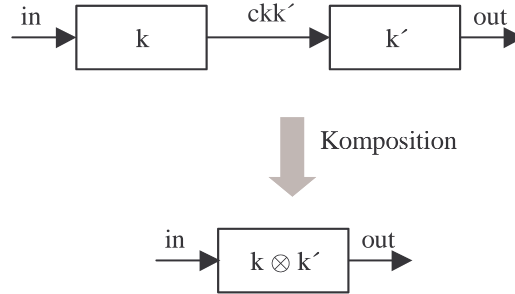


Abbildung 8.1: Strukturelle Aspekte der Komposition von Komponenten

Abbildung 8.1 stellt die strukturellen Aspekte der Komposition graphisch dar. Mathematisch definieren wir die Komposition wie folgt:

Definition 8.20 (Komposition von Komponenten)

Sei CH eine Menge typisierter Kanäle und seien $(I_1, O_1, F_1), (I_2, O_2, F_2) \in KP(CH)$ mit $I_1 \cap I_2 = \varnothing$ und $O_1 \cap O_2 = \varnothing$. Die Parallelkomposition mit Rückkopplung ⊗ definieren wir durch

$$(I_1, O_1, F_1) \otimes (I_2, O_2, F_2) =_{def} (I, O, F)$$

mit

$$I = (I_1 \cup I_2) \backslash (O_1 \cup O_2),\ O = (O_1 \cup O_2) \backslash (I_1 \cup I_2)$$

und $F \in \vec{I} \to \wp(\vec{O})$, wobei für alle $x \in \vec{I}$ gelte

$$F(x) = \{ y' \in \vec{O} \,|\, \exists y \in \overrightarrow{I_1 \cup I_2 \cup O_1 \cup O_2}.$$
$$y|_O = y' \wedge y|_I = x \wedge$$
$$y|_{O_1} = F_1(y|_{I_1}) \wedge y|_{O_2} = F_2(y|_{I_2}) \}$$

Dabei bezeichnen wir für jede Kanalbelegung x einer Kanalmenge C mit $x|_{C'}$ die Einschränkung der Belegung x auf die Kanäle in C´ (wobei $C' \subset C$ gelte).

□

Aufbauend auf dem Begriff der Komponente definieren wir *Komponenten-Netzwerke* als endliche Mengen von Komponenten, die durch *Punkt-zu-Punkt-Verbindungen* verbunden sind. Die syntaktische Schnittstelle eines Netzwerkes sowie das Verhalten bezüglich dieser Schnittstelle entspricht der Komponente, die wir erhalten, wenn wir die Komponenten des Netzwerkes gemäss Definition 8.20 komponieren:

Definition 8.21 (Komponentennetzwerke)

Sei CH eine Menge von Kanälen, dann definieren wir die Menge

$$NW(CH) \subseteq (\wp(KP(CH)) \backslash \varnothing)$$

der Komponenten-Netzwerke (bezüglich KP(CH) siehe Definition 8.19) durch

$$\begin{aligned} NW(CH) =_{def} \{ X \in \wp(KP(CH)) \mid \forall x, x' \in X, c \in CH. \\ (c \in in(x) \wedge c \in in(x') \Rightarrow x = x') \wedge \\ (c \in out(x) \wedge c \in out(x') \Rightarrow x = x') \} \end{aligned}$$

Wir schränken uns damit auf Netzwerke mit einer Punkt-zu-Punkt-Kommunikation ein.

Die syntaktische Schnittstelle eines Netzwerkes und das Netzwerkverhalten bezüglich dieser Schnittstelle ergibt sich unmittelbar aus der Komposition seiner Elemente. Für ein Netzwerk $nw \in NW(CH)$ mit $nw = \{c_1, c_2, ..., c_n\}, n \in \mathbb{N}$ gelte:

$$\begin{aligned} & in(nw) =_{def} \Pi_1(c') \\ & out(nw) =_{def} \Pi_2(c') \\ & behav(nw) =_{def} \Pi_3(c') \\ & \text{wobei } c' = c_1 \otimes c_2 \otimes ... \otimes c_n \end{aligned}$$

Mit in(nw) bezeichnen wir die *Eingabe*-, mit out(nw) die *Ausgabekanäle* und mit behav(nw) das *Verhalten eines Netzwerkes*. Die Menge aller Kanäle, also der sowohl der Schnittstellen- als auch der internen Kanäle, bezeichnen wir mit

$$channels(nw) =_{def} \bigcup_{c \in nw} in(c) \cup out(c)$$

□

Mit obiger Netzwerkdefinition haben wir das in FOCUS verwendete Modell hinsichtlich topologischer Aspekte erweitert:

In FOCUS können die Eigenschaften eines Systems oder einer Komponente durch ein Netzwerk interagierender Komponenten spezifiziert werden. Diese Netzwerkstruktur ist in FOCUS aber nur auf Ebene der (abstrakten) Syntax explizit. Bei der Abbildung in das semantische Modell wird davon abstrahiert, so dass auf semantischer Ebene (syntaktisch) unterschiedlich strukturierte Netzwerke, die dasselbe Verhalten beschreiben, gleichgesetzt werden.

In unserem Ansatz ordnen wir einer Netzwerkspezifikation als Semantik eine Menge von Netzwerken im Sinne von Definition 8.21 zu (vergleiche gegebenenfalls Definition 2.21). Durch ihre Komponenten enthalten Netzwerke Information über die Netzwerkstruktur/Topologie. So ordnen wir beispielsweise zwei Spezifikationen, die verhaltensäquivalente aber topologisch unterschiedliche Netzwerke beschreiben, auch unterschiedliche Netzwerke als Semantik zu. Indem wir strukturelle Information auch im semantischen Modell explizit machen, wird es beispielsweise möglich, zwischen logischen und technischen Netzwerkspezifikationen zu unterscheiden (vergleiche Definition 2.21), das heißt zwischen den Spezifikation die zwar in Form eines Netzwerkes gegeben sind, aber damit nur Schnittstelleneigenschaften und keine internen strukturellen Eigenschaften der Realisierung festlegen (logische Sicht) beziehungsweise Spezifikation, die auch die Topologie der Realisierung festlegen (technische Sicht, häufig auch als Deployment Sicht benannt, vergleiche etwa [JBR99]).

8.4.2 Interaktions-Zustands-Tupel und -Ströme

Zur Beschreibung der (getakteten) Ausführungsfolgen eines Systems, einer Komponente und von Automaten führen wir sogenannte *Interaktions-Zustands-Tupel* (wir schreiben im Folgenden kurz *IS-Tupel*) ein. Diese Form von 2-Tupeln setzen sich aus einem (Daten-)Zustand und einem Interaktionsmuster (gemäss Definition 8.18) zusammen. Dadurch stellen wir einen Bezug zwischen (Daten-)Zustand und Interaktionsverhalten, letzteres repräsentiert durch das Interaktionsmuster, her. Durch ein IS-Tupel (innerhalb eines Stroms von IS-Tupeln) modellieren wir den Zustand sowie die Interaktionen in einem bestimmten Takt/Zeitpunkt, so dass ein IS-Tupel als ein „Verhaltens-Schnappschuss" verstanden werden kann.

Ausführungsfolgen modellieren wir als Ströme über Interaktions-Zustands-Tupeln (synonym: *IS-Tupel* und *IS-Ströme*). Da wir in einem IS-Tupel sowohl den Zustand als auch das Interaktionsverhalten eines Systems betrachten, und da zwischen beiden Aspekten im allgemeinen eine Abhängigkeit besteht, ist ein Interleaving von zwei Strömen kein adäquates Mittel, um die Parallelkomposition von IS-Strömen zu beschreiben. Deshalb führen wir anschließend an die Definition von IS-Tupeln und IS-Strömen einen Operator zur Parallelkomposition ein, welcher die Eigenschaften von IS-Tupeln geeignet berücksichtigt, indem Zustände unifiziert und Interaktionen *taktweise* gemischt werden.

Definition 8.22 (Interaktions-Zustands-Tupel)

Sei CH eine endliche Menge typisierter Kanäle und S eine Menge von Zuständen. Wir definieren die Menge der Interaktions-Zustands-Tupel über CH und S durch

$$IS_{CH,S} =_{def} \vec{CH^*} \times S$$

Wir schreiben abkürzend IS für $IS_{CH,S}$ und IS^ω für $\left(IS_{CH,S}\right)^\omega$, wenn aus dem Kontext CH und S eindeutig hervor gehen.

□

Für die Definition der Parallelkomposition von Interaktions-Zustandsströmen benötigen wir die Hilfsfunktionen *filter* und *merge* und *merge**, welche der Partitionierung beziehungsweise dem Mischen von IS-Tupeln beziehungsweise von IS-Strömen dienen.

Definition 8.23 (Hilfsfunktionen *filter*, *merge* und *merge)**

Sei *CH* eine Menge von Kanälen und *N* eine Menge von Nachrichten, so dass CH mittels einer Abbildung ctype : CH → ℘(N) typisiert sind.

Um die vollständige, reihenfolgeerhaltende Partitionierung einer Sequenz in zwei Teilsequenzen zu beschreiben, definieren wir die Funktion

$$\text{filter} : \left(N^* \times \{0,1\}^\infty \times \{0,1\}\right) \rightarrow N^*$$

so dass für alle x, y ∈ N, xs ∈ N*, bs ∈ $\{0,1\}^\infty$ und b ∈ {0,1} gelte:

$$\text{filter}(\varepsilon, \text{bs}, \text{b}) =_{\text{def}} \varepsilon$$

$$\text{filter}(\text{x} \,\&\, \text{xs}, \text{y} \,\&\, \text{bs}, \text{b}) =_{\text{def}} \begin{cases} \text{x} \,\&\, \text{filter}(\text{xs}, \text{bs}, \text{b}) & \text{falls } \text{y} = \text{b} \\ \text{filter}(\text{xs}, \text{bs}, \text{b}) & \text{sonst} \end{cases}$$

Durch die Funktion

$$\text{merge} : \overrightarrow{\text{CH}^*} \times \overrightarrow{\text{CH}^*} \to \wp\left(\overrightarrow{\text{CH}^*}\right)$$

ordnen wir zwei Interaktionsmustern die Menge aller Interaktionsmuster zu, die sich durch reihenfolgeerhaltendes Mischen ergeben:

$$\begin{aligned} \text{merge}(\text{x}, \text{y}) =_{\text{def}} \Big\{ \text{r} \in \overrightarrow{\text{CH}^*} \,\Big|\, & \forall \text{c} \in \text{CH}. \exists \text{bs} \in \{0,1\}^{\infty}. \\ & \text{filter}(\text{r}(\text{c}), \text{bs}, 0) = \text{x}(\text{c}) \wedge \\ & \text{filter}(\text{r}(\text{c}), \text{bs}, 1) = \text{y}(\text{c}) \Big\} \end{aligned}$$

Die Funktion merge erweitern wir zu

$$\text{merge}^* : \left(\left(\overrightarrow{\text{CH}^*}\right)^{\omega} \times \left(\overrightarrow{\text{CH}^*}\right)^{\omega}\right) \to \wp\left(\left(\overrightarrow{\text{CH}^*}\right)^{\omega}\right)$$

um die „taktweise" Mischung zweier Ströme von Kanalbelegungen zu beschreiben. Für alle $\text{x}, \text{y} \in \overrightarrow{\text{CH}^*}, \text{xs}, \text{ys} \in \left(\overrightarrow{\text{CH}^*}\right)^{\omega}$ gelte:

$$\text{merge}^*(\varepsilon, \text{ys}) =_{\text{def}} \{\text{ys}\}$$
$$\text{merge}^*(\text{xs}, \varepsilon) =_{\text{def}} \{\text{xs}\}$$
$$\begin{aligned} \text{merge}^*(\text{x} \,\&\, \text{xs}, \text{y} \,\&\, \text{ys}) =_{\text{def}} \Big\{ \text{r} \in \left(\overrightarrow{\text{CH}^*}\right)^{\omega} \,\Big|\, & \text{ft}(\text{r}) \in \text{merge}(\text{x}, \text{y}) \wedge \\ & \text{rt}(\text{r}) \in \text{merge}^*(\text{xs}, \text{ys}) \Big\} \end{aligned}$$

□

Des weiteren benötigen wir die Projektionen auf den Zustands- und den Interaktionsanteil von IS-Strömen.

Definition 8.24 (Projektionen auf IS-Ströme)

Sei *CH* eine Menge von Kanälen und *N* eine Menge von Nachrichten, so dass CH mittels einer Abbildung ctype : CH → ℘(N) typisiert sind.

Für alle $\text{x} \in \text{IS}_{\text{CH,S}}, \text{xs} \in (\text{IS}_{\text{CH,S}})^{\omega}$ sei die Projektion auf den Zustandsanteil eines IS-Stroms

$$\Pi_{\text{Sstream}} : (\text{IS}_{\text{CH,S}})^{\omega} \to \text{S}^{\omega}$$

definiert durch

$$\Pi_{\text{Sstream}}(\varepsilon) =_{\text{def}} \varepsilon$$
$$\Pi_{\text{Sstream}}(\text{x} \,\&\, \text{xs}) =_{\text{def}} \Pi_2(\text{x}) \,\&\, \Pi_{\text{Sstream}}(\text{xs})$$

und analog dazu die Projektion auf den Interaktionsanteil

$$\Pi_{Istream} : (IS_{CH,S})^{\omega} \to \left(\vec{CH^*}\right)^{\omega}$$

definiert durch

$$\Pi_{Istream}(\varepsilon) =_{def} \varepsilon$$
$$\Pi_{Istream}(x \,\&\, xs) =_{def} \Pi_1(x) \,\&\, \Pi_{Istream}(xs)$$

□

Mit $\Pi_{Sstream}$ beziehungsweise $\Pi_{Istream}$ berechnen wir den in einem Interaktions-Zustands-Strom enthaltenen Zustands- beziehungsweise Interaktionsstrom.

Damit stehen uns alle Hilfsfunktionen zur Verfügung, um die Parallelkomposition von IS-Strömen definieren zu können.

Definition 8.25 (Parallelkomposition von IS-Strömen)

Für die Parallelkomposition von IS-Strömen

$$.\sim. : \left(IS^{\omega} \times IS^{\omega}\right) \to \wp\left(IS^{\omega}\right)$$

gelte für alle $x, y \in IS^{\omega}$:

$$x \sim y =_{def} \begin{cases} \varnothing & \text{falls } \neg((\Pi_S(x) \sqsubseteq \Pi_S(y)) \vee (\Pi_S(y) \sqsubseteq \Pi_S(x))) \\ \{r \in IS^{\omega} \mid \Pi_{Sstream}(r) = g(\Pi_{Sstream}(x), \Pi_{Sstream}(y)) \wedge \\ \qquad \Pi_{Istream}(r) \in merge*(\Pi_{Istream}(x), \Pi_{Istream}(y))\} \\ \text{sonst} \end{cases}$$

wobei

$$g : S^{\omega} \times S^{\omega} \rightharpoonup S^{\omega}$$

mit

$$g(x, y) =_{def} \begin{cases} x & \text{falls } y \sqsubseteq x \\ y & \text{falls } x \sqsubseteq y \end{cases}$$

für alle $x, y \in S^{\omega}$.

□

Definition 8.26 (Parallelkomposition von Mengen von IS-Strömen)

Durch elementweise Anwendung erweitern wir die Parallelkomposition von IS-Strömen auf Mengen von IS-Strömen:

Für $\sim * : \wp\left(IS^{\omega}\right) \times \wp\left(IS^{\omega}\right) \to \wp\left(IS^{\omega}\right)$ und alle $x, y \in \wp\left(IS^{\omega}\right)$ gelte

$$x \sim {}^*y =_{def} \left\{ \varphi \in IS^{\omega} \,\middle|\, \exists \varphi_x \in x, \varphi_y \in y . \varphi \in \left(\varphi_x \sim \varphi_y \right) \right\}.$$

Wenn durch den Anwendungskontext eindeutig festgelegt, schreiben wir ~ für ~*.

□

Folgende Vorstellung steht hinter der in Definition 8.25 gegebenen Parallel-Komposition:

Durch die Zustandskomponente eines IS-Tupels erfassen wir den *gesamten* System- beziehungsweise Komponentenzustand, und nicht nur einen Teil davon. Aus diesem Grund ergibt die Komposition zweier Verhalten nicht einen (Strom) komponierter Zustände. Vielmehr müssen zwei zu komponierende Verhalten bezüglich (des gemeinsamen Präfixes) ihrer Zustandsfolge übereinstimmen, so dass die Zustandsfolge des komponierten Verhaltens der Zustandsfolge der längeren der beiden zu komponierenden Verhalten entspricht. Gilt diese Übereinstimmung nicht, dann sind die zu komponierenden Verhalten nicht verträglich (da es mindestens einen Takt gibt, indem unterschiedliche Zustände gelten müssten), so dass die Parallelkomposition die leere Menge liefert.

Bezüglich der Folge von Interaktionsmustern eines IS-Stromes muss im komponierten Verhalten die relative Position jeder Kanalbelegung zum globalen Zustand erhalten bleiben, da im allgemeinen Interaktionen kausal für Zustandsänderungen sind, beispielsweise kann eine Nachricht *init* auf einem Eingabekanal einer Komponente *k* kausal für das Setzen des Komponentenzustandes auf einen Initialzustand sein.

Zudem definieren wir die Parallelkomposition so, dass alle Interaktionen der zu komponierenden Ströme im komponierten Verhalten enthalten sind. Gleichbenannte Interaktionen werden bei der Komposition *nicht* unifiziert. Dies ist für unseren Anwendungsfall der Parallelkomposition angemessen: Wir verwenden die Parallelkomposition, um die IS-Ströme zu erfassen, die sich durch die Realisierung kausal unabhängiger Prozessereignisse ergeben (siehe Abschnitt 2.6.2). Da jedes Prozessereignis *einem* „Auftreten" dessen entspricht, was wir mit der jeweiligen Ereignismarkierung assoziieren, beispielsweise eine Zustandsänderung, entsprechen mehrere, gleichmarkierte Ereignisse dem mehrfachen Auftreten. Ein Beispiel wäre die Realisierung zweier identisch markierter, kausal unabhängiger Überweisungen (siehe Abbildung 8.2).

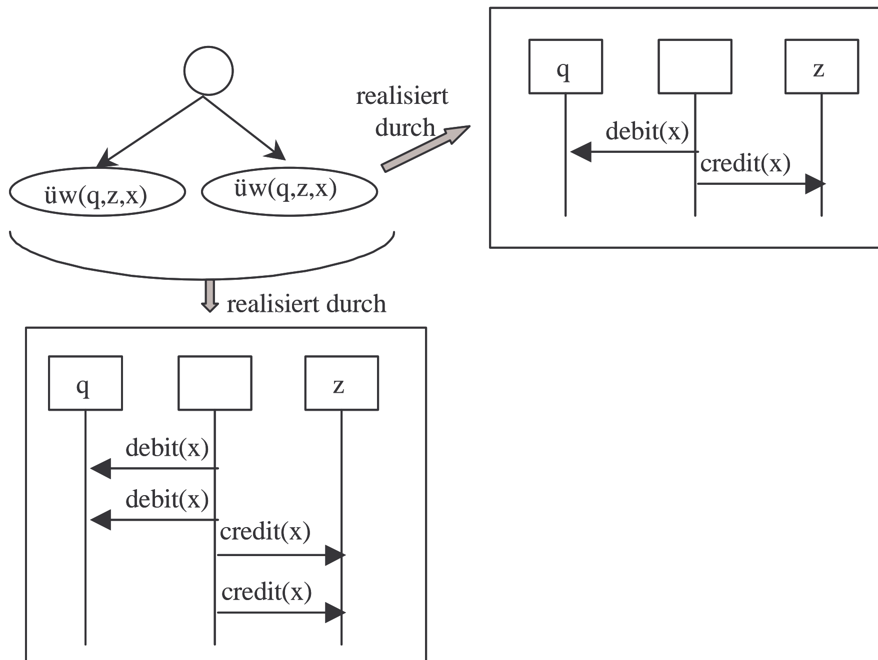


Abbildung 8.2: Realisierung *eines* Ereignisses und *eine* Alternative der Realisierung *zweier* identisch markierter, kausal unabhängiger Ereignisse.

8.4.3 Automaten

In diesem Abschnitt definieren wir eine Automatenart, die geeignet ist, um Verhalten von der Art von Komponenten zu beschreiben, die wir mit Definition 8.19 eingeführt haben. Diesem Komponentenbegriff entsprechend, assoziieren wir mit einer Automatentransition neben einem Zustandsübergang auch eine Ein- und Ausgabe. Eine Eingabe entspricht dabei einem Interaktionsmuster einer Menge von Eingabekanälen, eine Ausgabe einem Interaktionsmuster einer Menge von Ausgabekanälen:

Definition 8.27 (Automat)

Sei CH eine Menge typisierter Kanäle. Ein Automat ist ein Fünftupel $(S, I, O, \delta, Init)$ bestehend aus

- einer nichtleeren Menge von Zuständen S
- einer nichtleeren Menge (typisierter) Eingabekanäle $I \subseteq CH$
- einer nichtleeren Menge (typisierter) Ausgabekanäle $O \subseteq CH$
- einer Zustandsübergangsfunktion $\delta : \left(S \times \vec{I^*}\right) \rightarrow \wp\left(S \times \vec{O^*}\right)$ und
- einer nichtleeren Menge $Init \subseteq S \times \vec{O^*}$ von Paaren von Anfangszuständen und initialen Ausgaben.

Wir bezeichnen einen Automaten als *total*, falls gilt

$$\forall s \in S, i \in \vec{I^*}.\delta((s,i)) \neq \varnothing .$$

Die Totalität eines Automaten bedeutet, dass er *input enabled* ist (siehe [LT89]). Wird input enabledness von einem Automaten gefordert, so steht dahinter folgende Idee: Ein Automat hat keine Kontrolle über die Eingaben, die gemacht werden. Dennoch soll jedes Umgebungs-, das heißt Eingabeverhalten, möglich sein und nicht durch den Automaten blockiert werden. Dies entspricht der Vorstellung, dass Komponenten nur ihre Ausgaben, nicht aber ihre Eingaben kontrollieren (Ausnahme sind rückgekoppelte Kanäle).

Automaten sind eine Möglichkeit, um (Komponenten-)Verhalten zu beschreiben. Verhalten wird zum Beispiel durch Ausführungsfolgen (Spuren) oder Verhaltensfunktionen, die einen funktionalen Bezug zwischen Ein- und Ausgaben herstellen, modelliert. Automaten können bezüglich diesen Verhaltensmodellen auf unterschiedliche Weise interpretiert werden (siehe zum Beispiel [Bro97b]). In dieser Arbeit orientieren wir uns im Wesentlichen an der denotationellen Semantik der sogenannten Taktautomaten aus [Rum96].

Da wir in dieser Arbeit sowohl im Spur- als auch im funktionalen Verhaltensmodell einen gezeiteten/getakteten Verhaltensbegriff zugrunde legen, interpretieren wir Automaten so, dass sie ein gezeitetes Verhalten „generieren". Daher ergeben sich folgende Automateneigenschaften:

- pro Transition werden endliche Sequenzen von (Eingabe-)Nachrichten verarbeitet und endliche Nachrichtensequenzen ausgegeben (nicht nur einzelne Zeichen). Dies entspricht den Ein- beziehungsweise Ausgaben, die innerhalb eines Taktes gemacht werden können.
- Initialelemente sind Paare, bestehend aus einem Startzustand und einer für die erste Zeiteinheit bestimmten initialen Ausgabe.
- Pro Zeiteinheit wird genau eine Transition ausgeführt. Dabei wird die Eingabe dieser Transition, bestehend aus einer endlichen Sequenz von Nachrichten, verarbeitet und die Ausgabe dieser Transition in der nachfolgenden Zeiteinheit ausgegeben.
- Die Auswahl einer Transition erfolgt nichtdeterministisch aus der Menge der Transitionen, deren Ausgangszustand dem aktuellen Automatenzustand und deren Eingabe der aktuellen Automateneingabe entspricht.

Hinter dieser Automateninterpretation steht die Vorstellung, dass eine Transition einer atomaren Operation entspricht, die auf unteilbare Weise einen Zustandsübergang und eine Ausgabe erzeugt. Damit haben wir eine konzeptuell einfache Verhaltensinterpretation von Automaten gewählt, die Automaten für die implementierungsnahe Verhaltensspezifikationen geeignet macht.

Formal definieren wir das Automatenverhalten im Sinne von Ausführungsfolgen folgendermaßen:

Definition 8.28 (Ausführungsfolgen totaler Automaten)

Sei $(S, I, O, \delta, Init)$ ein Automat.

Wir ordnen jedem Zustand $s \in S$ eine Abbildung $\Delta s : \vec{I} \to \wp\left(S^{\infty} \times \vec{O}\right)$ zu. Wir fordern, dass die folgende Gleichung erfüllt ist:

$$\begin{aligned}
&\forall s \in S, i \in I^{*}, x \in \vec{I}. \\
&\Delta s(i \,\&\, x) = \Big\{(s' \& z, o \,\&\, y) \in \left(S^{\infty} \times \vec{O}\right) \Big| (s', o) \in \delta((s, i)) \wedge \\
&\qquad\qquad\qquad\qquad\qquad (z, y) \in \Delta s'(x)\Big\}
\end{aligned}$$

Zudem sei, für alle $s \in S$, Δs jeweils die, bezüglich der Mengeninklusion, größte Lösung.

Die Menge der (möglichen) Ausführungsfolgen eines totalen Automaten $(S, I, O, \delta, Init)$ definieren wir durch

$$\begin{aligned}
executions((S, I, O, \delta, Init)) = \Big\{ x \in \left(IS_{(I \cup O), S}\right)^{\infty} \Big| \;&\exists (s_0, o_0) \in Init. \\
&\Pi_1(ft(x))|_O = o_0 \wedge \\
&\Pi_2(ft(x)) = s_0 \wedge \\
&\left(\Pi_{Sstream}(rt(x)), \Pi_{Istream}(rt(x))|_O\right) \in \\
&\Delta s_0\left(\Pi_{Istream}(x)|_I\right)\Big\}
\end{aligned}$$

wobei $\Pi_{Sstream}, \Pi_{Istream}$ die in Definition 8.24 eingeführten Projektionen seien.

□

Wir interessieren uns für die, bezüglich der Mengeninklusion, größte Lösung für alle $s \in S, \Delta s$, da wir *alle* Ausführungsfolgen erfassen wollen, die durch ein schrittweises „Vorgehen/Verhalten", gemäss den durch die Funktion δ vorgegebenen „Verhaltensschritten", beobachtbar/möglich sind.

Eine Lösung der obigen, die Abbildungen $\Delta s, s \in S$, definierenden Gleichungen ist

$$\forall x \in \vec{I}, s \in S. \Delta s(x) = \varnothing .$$

Da wir totale Automaten betrachten, ist dies nicht die einzige Lösung, da

$$\forall i \in I^{*}, s \in S. \exists s', o. (s', o) \in \delta(s, i)$$

gilt, und somit das erste Glied der Konjunktion in der definierenden Gleichung erfüllbar ist.

Neben den Ausführungsfolgen dienen Automaten auch der Festlegung des Black-Box-Verhaltens von Komponenten. Das Black-Box-Verhalten einer Komponente modellieren wir, wie oben festgelegt, durch eine strikt kausale Verhaltensfunktion, die Eingabehistorien auf Mengen von Ausgabehistorien abbildet. Um die durch einen Automaten definierte Verhaltensfunktion zu bestimmen, gehen wir analog zu obiger Definition der Ausführungsfolgen eines Automaten vor. Der Unterschied besteht darin, dass von der Zustandsfolge abstrahiert wird, welche der „Berech-

nung“ einer Ausgabehistorie (zu einer gegebenen Eingabehistorie) zugrunde liegt. Zudem werden Ein- und Ausgabe in einen funktionalen Zusammenhang gestellt.

Definition 8.29 (Interaktionsverhalten totaler Automaten)

Sei $(S, I, O, \delta, Init)$ ein totaler Automat.

Wir ordnen jedem Zustand $s \in S$ eine Abbildung $[s] : \vec{I} \to \wp(\vec{O})$ zu. Wir fordern, dass die folgende Gleichung erfüllt ist:

$$\forall s \in S, i \in \vec{I^*}, x \in \vec{I}.$$
$$[s](i \,\&\, x) = \{o \,\&\, y \mid \exists s' \in S.((s', o) \in \delta((s, i))) \land (y \in [s'](x))\}$$

Zudem sei, für alle $s \in S$, $[s]$ jeweils die, bezüglich der Mengeninklusion, größte Lösung. Eine solche größte Lösung existiert, da der Automat total ist und somit für jede Eingabe, jede Ausgabesequenz ein Element, mindestens die leere Sequenz, enthält.

Das Interaktionsverhalten des totalen Automaten $(S, I, O, \delta, Init)$ modellieren wir durch eine Funktion der Funktionalität $\vec{I} \to \wp(\vec{O})$, die wir mit

$$[(S, I, O, \delta, Init)]$$

bezeichnen. Wir fordern für alle $x \in \vec{I}$:

$$\begin{aligned}[(S, I, O, \delta, Init)](x) = \Big\{ y \in \vec{O} \Big| \, & \exists (s_0, o_0) \in Init. \\ & ft(y) = o_0 \land \\ & (rt(y) \in [s_0](x)) \}\end{aligned}$$

□

Literaturverzeichnis

[ADH+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, A. Tarlecki: CASL: The Common Algebraic Specification Language. In: *Theoretical Computer Science*, 2002

[ADO00] W. van der Aalst, J. Desel, A. Oberweis (Hrsg.): *Business Process Management: Models, Techniques, and Empirical Studies*. Lecture Notes in Computer Science 1806, Springer, 2000

[AL90] M. Abadi, L. Lamport: *Composing Specifications*. Digital Systems Research Center, SRC Report 66, October 1990

[All97] R. Allen: *A Formal Approach to Software Architecture*. PhD Thesis, Carnegie Mellon, School of Computer Science, January 1997, Issued as CMU Technical Report CMU-CS-97-144

[ANS84] ANSI/IEEE: Std 830-1984.

[AS85] B. Alpern, F.B. Schneider: Defining Liveness. In: *Information Processing Letters*, 21, Elsevier, 1985, S.181-185

[ASU88] A. V. Aho, R. Sethi, J.D. Ullmann: *Compilerbau, Teil 1*. Addison Wesley, 1988

[BA81] J.D. Brock, W.B. Ackermann: Scenarios: A model of non-determinate computation. In J. Diaz, I. Ramos (Hrsg.): *Formalization of Programming Concepts*. Lecture Notes in Computer Science 107, Springer, 1981, S. 252-259

[Bal96] H. Balzert: *Lehrbuch der Software-Technik: Software-Entwicklung*. Spektrum Akad. Verlag, 1996

[BDI97] Bundesministerium des Inneren der Bundesrepublik Deutschland (Hrsg.): *Entwicklungsstandard für IT-Systeme der Bundes: Vorgehensmodell, Teil 1, Allgemeiner Umdruck 250/1*. Bonn, 1997

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stølen: *The Requirement and Design Specification Language SPECTRUM: An Informal Introduction*. Technischer Bericht TUM-I9311, Technische Universität München, 1993

[BM93] Ö. Babaoglu, K. Marzullo: Consistent Global States. In S. Mullender (Hrsg.): *Distributed Systems*. ACM Press, 1993

[BMP+86] M. Broy, B. Möller, P. Pepper, M. Wirsing: Algebraic implementations preserve program correctness. In: *Science of Computer Programming* 7, 1986, S.35-53

[Boe87] B. Boehm: A spiral model of software development and enhancement. In: *Software Engineering Project Management*, 1987, S. 128-142

[Box98] D. Box: *Essential COM - (The DevelopMentor Series).* Addison Wesley, 1998

[BP00] Max Breitling, Jan Philipps: Step by Step to Histories. In Theodor Rus (Hrsg.): *Proceedings of AMAST 2000, Algebraic Methodology and Software Technology Iowa City, IA*. Lecture Notes in Computer Science 1816, Springer, 2000, S. 11-25

[Bre91] R. Breu: *Algebraic Specification Techniques in Object Oriented Programming Environments*. Lecture Notes in Computer Science 562, Springer, 1991

[Bre98] R. Breu: *Konzepte, Techniken und Methodik des objektorientierten Entwurfs – Ein integrierter Ansatz.* Habilitationsschrift, Technische Universität München, 1998

[Bro01] M. Broy: *Closure Operations on Incomplete State Transition Diagrams*. Internes Papier, 2001

[Bro89] M. Broy: Towards a Design Methodology for Distributed Systems. In M. Broy (Hrsg.): *Constructive Methods in Computer Science*. Int. Summer School, NATO ASI Series, Series F: Computer and Systems Sciences, Volume 55, Springer, 1989, S.311-364

[Bro92] M. Broy: Algebraic and functional specification of an interactive serializable database interface. In: *Distributed Computing*, 6, 1992, S. 5-18

[Bro94] M. Broy: *A Functional Rephrasing of the Assumption/Commitment Specification Style*. Technischer Bericht TUM-I9417, Technische Universität München, 1994

[Bro95] M. Broy: Mathematical System Models as a Basis of Software Engineering. In J. van Leeuwen (Hrsg.): *Computer Science Today, Recent Trends and Developments*. Lecture Notes in Computer Science 1000, Springer, 1995, S.292-306

[Bro97a] M. Broy: Requirements engineering for embedded systems. In: *Proceedings of FemSys'97, München*. April 1997

[Bro97b] M. Broy: *The Specification of System Components by State Transition Diagrams*. Technischer Bericht TUM-I9729, Technische Universität München, 1997

[BS00] M. Broy and K. Stølen: *Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement*. Springer, 2000

[BSI00] Bundesamt für Sicherheit in der Informationstechnik - BSI (Hrsg.): *Kommunikations- und Informationstechnik 2010, Trends in Technologie und Markt*. Bonn, 2000

[Bud97] T. Budd: *An Introduction to Object-Oriented Programming*. 2nd Edition. Addison Wesley, 1997

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes: *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994

[CD94] S. Cook, J. Daniels: *Designing Object Systems – Object-Oriented Modelling with Syntropy*. Prentice Hall, 1994

[Che76] P. Chen: The entity-relationship model – towards a unified view of data. In: *ACM Transactions on Database Systems*, Vol. 1, No. 1, 1976, S.9-36

[CM88] K. M. Chandy, J. Misra: *Parallel Program Design, A Foundation*. Addison-Wesley, 1988

[CY91] P. Coad, E. Yourdon: *Object-Oriented Analysis*. 2nd ed.. Prentice Hall, 1991

[Dav93] Th. H. Davenport: *Process Innovation – Reengineering Work through Information Technology*. Harvard Business School Press, 1993

[DAW99] J.-C. Derniame, B. Ali Kaba, D. Wastell (Hrsg.): *Software Process, Principles, Methodology, and Technology*. Lecture Notes in Computer Science 1500, Springer, 1999

[DCC92] E. Down, I. Coe, P. Clare: *Structured systems analysis and design method: application and context*. 2nd ed., Prentice Hall, 1992

[Dei01] B. Deifel: *Requirements Engineering komplexer Standardsoftware*. Dissertation, Technische Universität München, 2001

[DSV99] B. Deifel, W. Schwerin, S. Vogel : *Work Products for Integrated Software Development*. Technischer Bericht TUM-I9921, Technische Universität München, Dezember 1999

[DT90] M. Dorfman, R.H. Thayer: *Standards, Guidelines, and Examples on System and Software Requirements Engineering*. IEEE Computer Society Press Tutorial, IEEE Computer Society Press, 1990

[DW99a] Desmond F. D'Souza, Alan Cameron Wills: *Objects, Components and Frameworks with UML; the Catalysis Approach*. Addison-Wesley Object Technology Series, 1999

[EKM+82] H. Ehrig, H.-J. Kreowski, B. Mahr, P. Padawitz: Algebraic implementation of abstract data types. In: *Theoretical Computer Science* 20, 1982, S. 209-263

[Eme90] E.A. Emerson: Temporal and Modal Logic. In J. van Leeuwen (Hrsg.): *Handbook of Theoretical Computer Science*. Volume B. Formal Models and Semantics. Elsevier, 1990, S.995-1072

[Fran86] N. Francez: *Fairness*. Springer, 1986

[GG98] R. van Glabbeek, U. Goltz: *Refinement of Actions and Equivalence Notions for Concurrent Systems*. Hildesheimer Informatik Bericht 6/98, Institut für Informatik, Universität Hildesheim, 1998

[GH93] J.V. Guttag, J.J. Horning: *Larch: Languages and Tools for Formal Verification*. Texts and Monographs in Computer Science, Springer, 1993

[GHJ+95] E. Gamma, R. Helm, R. Johnson, J.Vlissides: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995

[Gro98] The Object Management Group (Hrsg.): *Common Object request broker architecture*. 1998

[GS87] H. Garcia-Molina, K. Salem: “Sagas”. In: *Proceedings of the ACM SIGMOD Int. Conf. on the Management of Data*, 1987

[GTW78] J.A. Goguen, J.W. Thatcher, E.G. Wagner: *An initial algebra approach to the specification correctness, and implementation of abstract data types*. IBM Research Report RC-6487; auch in: R.T. Yeh (Hrsg.): Current Trends in Programming Methodology, Vol.4: Data Structuring. Prentice Hall, 1978, S.80-149

[Har97] D. Harel: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* (8), Elsevier, 1987

[Het95] R. Hettler: *Entity/Relationship-Datenmodellierung in axiomatischen Spezifikationssprachen*. Dissertation, Technische Universität München, 1995

[Hin98] U. Hinkel: *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL*. Dissertation, Technische Universität München, 1998

[HNS99] C. Hofmeister, R. Nord, D. Soni: *Applied Software Architecture*. Addison Wesley Longman, 1999

[Hoa72] C. A. R. Hoare: Proof of Correctness of Data Abstractions. In: *Acta Informatica* 1, Springer, 1972, S. 271-281

[Hoa85] C. A. R. Hoare: *Communicating Sequential Processes*. Prentice-Hall, International Series in Computer Science, 1985

[HU94] J.E. Hopcroft, J. Ullman: *Introduction to automata theory, languages and computation*. Addison-Wesley, 1994

[Huß93a] H. Hußmann: *Synergy between Formal and Pragmatic Software Engineering Methods*. Technischer Bericht TUM-I9323, Technische Universität München, 1993

[Huß93b] H. Hußmann: *Zur formalen Beschreibung der funktionalen Anforderungen an ein Informationssystem*. Technischer Bericht TUM-I9332, Technische Universität München, 1993

[Huß97] H. Hußmann: *Formal Foundations for Software Engineering Methods*. Lecture Notes in Computer Science 1322, G. Goos, J. Hartmanis, J. van Leeuwen (Hrsg.), Springer, 1997

[IBM97] IBM Object-Oriented Technology Center: *Developing object-oriented software: an experience-based approach*. Prentice Hall, 1997

[ISO95] International Standard, Information Technology, Software Lifecycle Process. 12207, 1995

[Jac01] M. Jackson: *Problem Frames, Analyzing and structuring software development problems*. Addison Wesley, ACM Press, 2001

[Jac92] I. Jacobson: *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, 1992

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*. Addison Wesley Longman, 1999

[JEJ94] I. Jacobson, M. Ericsson, A. Jacobson : *The Object Advantage - Business Process Reengineering With Object Technology*. Addison-Wesley, 1994

[JVW00] G.K. Janssens, J. Verelst, B. Weyn: Techniques for Modelling Workflows and Their Support of Reuse. In: W. van der Aalst, J. Desel, A. Oberweis (Hrsg.): *Business process management: models, techniques and empirical studies*. Lecture Notes in Computer Science 1806, Springer, 2000, S.1-29

[Kah74] G. Kahn: The semantics of a simple language for parallel programming. In: *Proceedings of Information Processing* 74, North-Holland, 1974

[Kle97] C. Klein: *Anforderungsspezifikation durch Transitionssysteme und Szenarien*. Dissertation, Technische Universität München, 1997

[KM77] G. Kahn, D.B. MacQueen: Coroutines and networks of parallel processes. In: *Proceedings of Information Processing* 77, North-Holland, 1977, S.993-998

[Kos62] E. Kosiol: *Organisation der Unternehmung*. Gabler Verlag, 1962

[Krü01] I. Krüger: *Distributed System Design with Message Sequence Charts*. Dissertation, Technische Universität München, 2001

[Kuv94] P. Kuvaja: *Software process assessment and improvement. The bootstrap approach*. Blackwell, Oxford, UK, 1994

[Lam77] L. Lamport: Proving the correctness of multiprocess programs. In: *IEEE Transactions Software Engineering SE-3* (2), 1977, S.125-143

[Lam80] L. Lamport: “Sometime” is Sometimes “Not Never”. In: *Proceedings of the 7th Annual ACM Sicact-Sigplan Symposium on Principles of Programming Languages*, New York. ACM Press, 1980, S.174-185

[Lam94] L. Lamport: The Temporal Logic of Actions. In: *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, May 1994, S.872-923

[LS84] J. Loeckx, K. Sieber: *The Foundations of Program Verification*. John Wiley & Sons and Teubner, Stuttgart, 1984.

[LSW01] M.C. Little, S.K. Shrivastava, S.M. Wheater: *A framework for implementing extended transactions*. Computing Science Department, University of Newcastle, Newcastle upon Tyne, England, 2001

[LSW97] P. Langner, P. Schneider, C. Wehler: *Prozessmodellierung mit ereignisgesteuerten Prozessketten (EPKs) und Petri-Netzen*. Wirtschaftsinformatik 39, Institut für Wirtschaftsinformatik, Universität des Saarlandes, 1997, S.479-489

[LT89] N. Lynch, M. Tuttle: An Introduction to Input/Output automata. In: *CWI-Quarterly*, 2, Nr.3, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, September 1989, S. 219-246

[Mil89] R. Milner: *Communication and Concurrency*. Prentice Hall, 1989

[MP92] Z. Manna, A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992

[Mul93] S. Mullender: *Distributed Systems*. ACM Press, 1993

[NKF94] B. Nuseibeh, J. Kramer, A. Finkelstein: A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. In: *IEEE Transactions on Software Engineering*, 20, Nr.10, Oct. 1994, IEEE, 1994, S. 760-773

[Nüt98] M. Nüttgens, T. Feld, V. Zimmermann: Business Process Modeling with EPC and UML: Transformation or Integration ?. In M. Schader, A. Korthaus (Hrsg.): *The Unified Modeling Language*. Physica Verlag, 1998

[OHM+91] T.W. Olle, J. Hagelstein, I.G. Macdonald, C. Rolland, H.G. Sol, F.J.M. van Assche, A.A. Verjin-Stuart: *Information Systems Methodologies*. Addison-Wesley, 1991

[OMG01] OMG: *Unified Modeling Language Specification, Version 1.4*. September 2001, http://www.omg.org/

[Öst95a] H. Österle: *Business Engineering, Prozeß- und Systementwicklung, Band 1 Entwurfstechniken*. Springer, 1995

[Pae98a] B. Paech: The Four Levels of Use Case Description. In E. Dubois, A. Opdahl, K. Pohl (Hrsg.): *4th Int. Workshop on Requirements Engineering: Foundations for Software Quality*, Pisa, June1998

[Pae98b] B. Paech: *Aufgabenorientierte Softwareentwicklung, Modellierung und Gestaltung von Unternehmen, Arbeit und Software*. Habilitationsschrift, Technische Universität München, 1998

[Pan90] P.K. Pandyà: Some Comments on the Assumption-Commitment Framework for Compositional Verification of Distributed Programs. In J.W. de Bakker, W.-P. de Roever, G. Rozenberg (Eds): *Stepwise Refinement of Distributed Systems*. Lecture Notes in Computer Science 430, Springer, 1990, S.622-640

[Par83] D. Park: The "fairness" problem and nondeterministic computing networks. In: *Proceedings of the 4th Foundations of Computer Science*, Number 159 in Mathematical Centre Tracts, Mathematisch Centrum Amsterdam, 1983, S. 133-161

[PCC+93] M.C. Paulk, B. Curtis, M.B. Chrissis, C.V. Weber: *Capability Maturity Model for Software, Version 1.1*. Technical Report, CMU/SEI-93-TR-024. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1993

[Poh96] K. Pohl: *Process-Centered Requirements Engineering*. Research Studies Press Ltd, JohnWiley & Sons Inc., 1996

[Pra86] V. Pratt: Modelling Concurrency with Partial Orders. In: *International Journal of Parallel Programming*, 15 (1), Nov. 1986, S.33-71

[PTL+99] S.J. Prowell, C.J. Trammell, R.C. Linger, J. H. Poore: *Cleanroom Software Engineering, Technology and Process*. Addison Wesley Longman, Inc., 1999

[PWG+93] M. C. Paulk, C. V. Weber, S. M. Garcia, M. B. Chrissis, M. Bush: *Key Practices of the Capability Maturity Model, Version 1.1*. Technical Report, CMU/SEI-93-TR-025, ESC-TR-93-178, February 1993

[Rei95] W. Reisig: Petri Net Models of Distributed Algorithms. In J. van Leeuwen (Hrsg.): *Computer Science Today, Recent Trends and Developments*. Lecture Notes in Computer Science 1000, Springer, 1995, S.441-454

[RG94] C. Rolland, G. Grosz: A General Framework for Describing the Requirements Engineering Process. In: *Proceedings of the Int, Conf. on Systems, Man and Cybernetics, San Antonio, TX, October 1994*. IEEE Computer Society Press, 1994

[Rou95] T. Rout: SPICE: A Framework for Software Process Assessment. In: *Software Processes: Improvement and Practice*, 1, Nr.1, 1995

[Roy70] W.W. Royce: Managing the development of large software systems. In: *Tutorial: Software Engineering Project Management*. IEEE Computer Society, Washington D.C., S. 188-127, 1970

[Rum96] Bernhard Rumpe: *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996

[Sal02] C. Salzmann: *Modellbasierter Entwurf spontaner Komponentensysteme*. Dissertation, Technische Universität München, 2002

[Schee96] A.-W. Scheer: *ARIS-House of Business Engineering*. Heft 133, Institut für Wirtschaftsinformatik, Universität des Saarlandes, Saarbrücken, September 1996

[Schee98] A.-W. Scheer: *ARIS, Modellierungsmethoden, Metamodelle, Anwendungen*. Springer, 1998

[Schr93] M. D. Schroeder: A State-of-the-Art Distributed System: Computing with BOB. In S. Mullender: *Distributed Systems*. ACM Press, 1993

[Schw00] W. Schwerin: Models of Systems, Work Products, and Notations. In: *Proceedings of Intl. Workshop on Model Engineering, ECOOP'2000, Cannes France*. June 2000

[SGW94] B. Selic, G. Gullekson, P.T. Ward: *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994

[SHB96] B. Schätz, H. Hußmann, M. Broy: Graphical Development of Consistent System Specifications. In J. Woodcock, M.-C. Gaudel (Hrsg.): *FME´96: Industrial Benefit and Advances in Formal Methods*. Lecture Notes in Computer Science 1051, Springer, 1996

[SS01] C. Salzmann, W. Schwerin: *Logical and technical abstractions for software architecture*. Technischer Bericht Arbeitspapier, Technische Universität München, 2001

[Stau01] T. Stauner: *Systematic Development of Hybrid Systems. Dissertation*. Technische Universität München, 2001

[Sun95] Sun Microsystems: *The Java Language Specification*. Sun Microsystems, Mountain View, CA, 1995

[Tan92] A.S. Tanenbaum: *Modern Operating Systems*. Prentice Hall, 1992

[TWW82] J.W. Thatcher, E.G. Wagner, J.B. Wright: Data type specification: parameterization and the power of specification techniques. In: *ACM TOPLAS*, 4, 1982, S.711-773

[Web91] R. Weber: *Eine Methodik für die formale Anforderungsspezifikation verteilter Systeme*. Dissertation, Technische Universität München, 1991

[WFMC99] Workflow Management Coalition: *Terminology & Glossary*. Document Number WFMC-TC-1011, Status 3, www.wfmc.org, February 1999

[Win82] G. Winskel: Event Structure Semantics for CCS and Related Languages. In G. Goos, J. Hartmanns (Hrsg.): *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 1982*. Lecture Notes in Computer Science 140, Springer, 1982, S. 561-576

[Win87b] G. Winskel: Event Structures. In: *Petri Nets: Applications and Relationships to other Models of Concurrency*. Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, September 1986, Lecture Notes in Computer Science 255, Springer, 1987, S. 325-392

[Wir71] N. Wirth: Program development by stepwise refinement. In: *Communications of the ACM*, Vol. 14, No. 4, 1971, S.221-227

[Wir90] M. Wirsing: Algebraic Specification. In J. van Leeuwen (Hrsg.): *Handbook of Theoretical Computer Science. Volume B. Formal Models and Semantics*. Elsevier, 1990, S.675-788

[WPP+83] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, M. Broy: On Hierarchies of Abstract Data Types. In F.L. Bauer, E.W. Dijkstra, A. Ershov, D. Gries, A.J. Perlis, W.M. Turski: *Acta Informatica*, Vol.20, Fasc.1, Springer 1983, S.1-34

[ZJ93] P. Zave, M. Jackson: Conjunction as Composition. In: *ACM Transactions on Software Engineering and Methodology*, Vol. 2, No. 4, 1993, S.379-411

[ZJ97] P. Zave, M. Jackson: Four Dark Corners of Requirements Engineering. In: *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No.1, 1997, S.1-30