Inhaltsverzeichnis

TEIL I PROLOG 1

1 EINLEITUNG 3
1.1 Erwartungen an die komponentenbasierte Softwareentwicklung 4
1.2 Ein Gestaltungsprozess für komponentenbasierte Anwendungen 6
1.3 Modelle/Beschreibungen von Komponenten und Anwendungssystemen 9
1.4 Hypothesen der Arbeit 11
1.5 Zielsetzung der Arbeit 12
1.6 Gliederung der Arbeit 15
2 BETRIEBLICHE ANWENDUNGSSYSTEME 18
2.1 Grundlagen 18
2.2 Konfiguration von Standardsoftware 24
2.3 Konfigurationshilfsmittel 28
3 KOMPONENTENBASIERTE ANWENDUNGSSYSTEME 33
3.1 Grundlagen 33
3.2 Komponenten 37
3.3 Komponentenschnittstellen 48
3.4 Softwarekonfigurationsmanagement 54
3.5 Fazit 58
4 SPRACHEN ZUR VERHALTENSBESCHREIBUNG 60
4.1 Kategorien von Verhaltensbeschreibungen 61
4.2 Sprachen zur Geschäftsprozessmodellierung 71
4.3 Sprachen zur objektorientierten Softwareentwicklung 77
4.4 Sprachen zur Beschreibung von Softwarearchitekturen 83
4.5 Einordnung der Sprachen zur Verhaltensbeschreibung 87
4.6 Fazit 88
5 NORMSPRACHEN 89
5.1 Formale versus natürliche Sprache 89
5.2 Einschub: Interpretation des Customizing-Prozesses 93
5.3 Gegenstandseinteilung 94
5.4 Wortarten 96
5.5 Elementarsätze 97
5.6 Bezug zur formalen Logik 100
5.7 Komplexe Satzarten von Normsprachen 101
5.8 Grammatik 101
5.9 Termini, Terminologie, Definition 103
5.10 Beschreibung kausaler Zusammenhänge 104
5.11 Fazit 105

TEIL II PROZESSORIENTIERTE KONFIGURATION VON ANWENDUNGSSYSTEMEN 107

6 KONKRETISIERUNG DER PROBLEMSTELLUNG 109
6.1 Anforderungen des Anwendungsarchitekten 109
6.2 Anforderungen des Konfigurierers 110
6.3 Anforderungen des Anwenders/Benutzers 111
6.4 Voraussetzungen der Komponententechnologie 111
6.5 Fazit 116
7 EIN EINFACHES KOMPONENTENMODELL 118
7.1 Ein SCDL-Beispiel 120
7.2 Syntax von SCDL 124
7.3 Semantik von SCDL 127
7.4 Formale Traces 139
8 NORMSPRACHEN UND KOMPONENTEN 143
8.1 Normsprachliche Kommunikation zwischen Komponenten 144
8.2 Interpretation der Komponentenbeschreibungen 147
8.3 Annotationen für Komponenten 155
9 ANWENDUNGSSOFTWAREPRODUKTE 156
9.1 Begriffe der Konfiguration 157
9.2 Konfiguration von Softwarekomponenten 159
9.3 Die Konfigurationsbeschreibungssprache CCL 164
10 GESCHÄFTSPROZESSORIENTIERTE KONFIGURATION 179
10.1 Anforderungsvalidierung und -verifikation 180
10.2 Zusammenhang von Konfiguration und Anforderungen 182
10.3 Ableitung von Geschäftsprozessen 188
10.4 Algorithmus zur prozessorientierten Konfiguration 189
10.5 Fazit 193

TEIL III IMPLEMENTIERUNG VON PROSECCO 195

11 ARCHITEKTUR 197
11.1 Architekturübersicht 197
11.2 Anwendungsebene 198
11.3 Persistenzebene 201
11.4 Benutzungsebene 202
12 IMPLEMENTIERUNGSASPEKTE 203
12.1 Anwendungsebene 203
12.2 Persistenzebene 209
12.3 Benutzungsebene 209

TEIL IV EPILOG 211

13 ZUSAMMENFASSUNG UND AUSBLICK 212
14 ABBILDUNGSVERZEICHNIS 215
15 TABELLENVERZEICHNIS 217
16 ZEICHENERKLÄRUNGEN 218
16.1 Abkürzungen 218
16.2 EBNF 219
LITERATURVERZEICHNIS 221

TEIL V ANHANG 229

17 DER π-KALKÜL 231
17.1 Synchroner π-Kalkül 231
17.2 Pict 233
18 BESCHREIBUNGSLOGIK 245
18.1 Der terminologische Formalismus 245
18.2 Abstrakte Syntax von Beschreibungslogiken 247
18.3 Semantik von Beschreibungslogiken 249
GLOSSAR 251
INDEX 261

Teil I Prolog

1 Einleitung

Betriebliche Anwendungssysteme werden heute maßgeblich durch integrierte Standardsoftwareprodukte wie das R/3-System geprägt. Doch es ist absehbar, dass diese sehr großen, monolithischen Systeme zugunsten kleinerer, flexiblerer Lösungen nach und nach aufgebrochen werden. Dazu werden sich vermehrt Standards etablieren, die es kleineren Herstellern ermöglichen werden, ihre Speziallösungen als Komponenten in die dann als Rahmenwerke (Frameworks) ausgelegten Systeme der großen Softwarehersteller wie SAP und Oracle einzubinden.

Für den Anwender bieten sich durch diese Öffnung gegenüber kleineren Speziallösungen zunächst eine größere Unabhängigkeit und Flexibilität bei der Gestaltung seiner Anwendungssysteme. Für die Gestaltungspotentiale betrieblicher Anwendungssysteme ist dieser Trend erst einmal positiv zu bewerten, da der Anwender sich nicht mehr en bloc für ein Gesamtpaket entscheiden muss, sondern die Möglichkeit hat, aus einer Menge alternativer Lösungen die besten auszuwählen. Allerdings haben vor allem kleine und mittelständische Unternehmen heute schon große Probleme, die Komplexität der Einführung eines Standardsoftwareprodukts, insbesondere der Konfiguration („Customizing“) dieser Systeme (siehe hierzu [Appelrath und Ritter 1999]), zu bewerkstelligen. Durch die zusätzliche Zerstückelung („Componentization“) zukünftiger Softwareprodukte steht der Anwender dann vor einer schier unüberschaubaren Vielfalt an Softwarekomponenten sowie Kombinations- und Konfigurationsmöglichkeiten dieser Komponenten.

Aus diesen Gründen werden spezielle Methoden und Werkzeuge benötigt (siehe [Ritter 1998a]), die den Anwender bei Auswahl, Konfiguration und Einsatz von betrieblichen Softwarekomponenten unterstützen.

Grundlegende Idee dieser Arbeit ist es, die Bemühungen, große betriebliche Softwareprodukte wie das R/3-System durch strukturierte, semi-formale Beschreibungen („Softwarereferenzmodelle“) für den Anwender verständlicher zu machen, auf die in Zukunft vermehrt anzutreffenden komponentenbasierten Softwareprodukte zu übertragen: Statt wie bisher ein Softwarereferenzmodell analog zu dem beschriebenen Produkt „in einem Guss“ zu liefern, sollen dann kleine Softwarereferenzmodelle der Komponenten erstellt werden, die – wie die Komponenten selbst – zu großen Systemen/Softwarereferenzmodellen zusammengesetzt werden können (siehe [Ritter 1998b]). So können Anwender, aber auch andere an der Gestaltung betrieblicher Anwendungssysteme beteiligte Akteure wie Entwickler und Berater sich frühzeitig einen Eindruck von den zukünftigen Systemen verschaffen.

In diesem Zusammenhang werden u.a. folgende, typische Fragestellungen behandelt:

- Wie können Softwarekomponenten und daraus abgeleitete Anwendungssysteme beschrieben werden, damit sie den Ansprüchen der an dem Gestaltungsprozess beteiligten Akteure gerecht werden?
- Wie können Komponenten und ihre Konfigurationen vor dem Hintergrund der Austauschbarkeit durch Konkurrenzprodukte anderer Hersteller beschrieben werden?
- Wie kann aus den möglichen Konfigurationen von Komponenten für den Anwender möglichst einfach eine geeignete Konfiguration ausgewählt werden?

Bevor konkreter auf diese Fragestellungen eingegangen wird, sollen zunächst die dieser Arbeit zugrunde liegenden Erwartungen an die komponentenbasierte Softwareentwicklung erörtert werden (Abschnitt 1.1). Darauf aufbauend wird in Abschnitt 1.2 ein Gestaltungsprozess vorgestellt, der ein denkbares Zusammenspiel der Softwarehersteller, Berater und Anwender skizziert, die an der Einführung betrieblicher Anwendungssysteme beteiligt sind. Danach werden in Abschnitt 1.3 einige wichtige Aspekte von Beschreibungssprachen für Softwarereferenzmodelle erörtert. In Abschnitt 1.4 werden die Haupthypothesen, auf denen diese Arbeit beruht, vorgestellt. In Abschnitt 1.5 wird die Zielsetzung dieser Arbeit genauer beschrieben. Zum Schluss dieses Kapitels wird in Abschnitt 1.6 ein Überblick über die weiteren Kapitel dieser Arbeit gegeben.

1.1 Erwartungen an die komponentenbasierte Softwareentwicklung

Eines der zentralen Ziele des Software Engineerings ist die Beschleunigung des Gestaltungsprozesses von (betrieblichen) Anwendungssystemen. Neben der Verbesserung des Softwareentwicklungsprozesses und dessen zunehmender Werkzeugunterstützung spielt das Gebiet der *Wiederverwendung* von Softwareelementen eine wesentliche Rolle. Prinzipiell gilt es dabei die folgenden Ansätze der Wiederverwendung zu unterscheiden:

1. Wiederverwendung durch Nutzung von Klassenbibliotheken und -Frameworks: *Klassenbibliotheken* und *-Frameworks* sind Softwarezwischenprodukte, die von Entwicklern genutzt werden können, um (schneller) größere Anwendungen zu entwickeln. Ähnliche Zwischenprodukte sind Templates und Makros.

2. Wiederverwendung durch Etablierung und Nutzung von Analyse- und Entwurfswissen: Analyse- (siehe [Fowler 1997]) und Entwurfsmuster (siehe [Gamma et al. 1995]) sind typische Ansätze für die systematische Wiederverwendung im Rahmen der frühen Phasen eines Entwicklungsprozesses.

3. Wiederverwendung mittels generativer Programmierung: Bei der generativen Modellierung wird zuerst ein möglichst abstraktes, aber ausreichendes Modell eines Anwendungsbereiches erstellt, aus dem dann mittels Programmfragmenten (Komponenten, Templates usw.) eine konkrete Implementierung generiert wird.

4. Wiederverwendung auf Basis konfigurierbarer Softwareprodukte: Konfigurierbare Softwareprodukte sind von Seiten des Softwareherstellers vollständig entwickelt, müssen jedoch beim Anwender noch auf seine speziellen Anforderungen eingestellt werden.

5. Wiederverwendung auf Basis von Komponenten: Komponenten sind in sich geschlossene, meist in binärer Form vorliegende Softwareprodukte (Black Boxes), die auf eine bestehende Softwarearchitektur, sogenannte *Komponenten-Frameworks*, abgestimmt sind und dort eingesetzt werden können (Plug-Ins). Komponenten werden von den Anwendern oder sie unterstützenden Dritten (Konfigurierern) zu vollständigen Anwendungssystemen zusammengesetzt.

In dieser Arbeit steht der Aspekt der Wiederverwendung auf Basis von Komponenten (5. Ansatz) im Vordergrund.

Dabei werden hauptsächlich *Anwendungskomponenten* betrachtet, also Komponenten, die sich in einer typischen Drei-Ebenen-Architektur zwischen den (Präsentations-)Clients und den Datenbankservern befinden und für die *Anwendungslogik* eines betrieblichen Anwendungssystems zuständig sind.

Bei allen oben angeführten Ansätzen erhofft man sich eine gesteigerte Softwarequalität und eine Reduzierung von Kosten, Risiken und Aufwand. Softwarequalität (siehe [Pomberger und Blaschek 1993]) zeigt sich in der Korrektheit, Zuverlässigkeit, Benutzer- und Wartungsfreundlichkeit sowie der Effizienz und Portabilität der Software. Eine Reduzierung von Kosten, Risiken und Aufwand erhofft man sich vor allem auch dadurch, dass Zwischenprodukte der Softwareentwicklung standardisiert und damit mehrfach verwendet werden können. Ein Problem ist jedoch, dass diese Zwischenprodukte von denjenigen, die sie einsetzten sollen, erst verstanden werden müssen. Diese und andere Barrieren haben in der Vergangenheit oft dazu geführt, dass Entwickler doch wieder „from scratch“ alles eigenständig entwickelt haben.

Vorteil der letzten drei Wiederverwendungsansätze ist, dass die darauf basierende Anwendungssystemgestaltung in die Hände des Anwenders gegeben werden kann. Es werden – je nach Qualität des Ansatzes – keine Softwareentwickler mehr benötigt, die die wiederzuverwendenden Zwischenprodukte zu einem Ganzen zusammenfügen. Typische Beispiele für Ansatz 4 sind die im betrieblichen Umfeld vorherrschenden ERP-Systeme wie das R/3-System von der SAP. Diese Systeme setzen auf einem zentralen Datenbankmanagementsystem auf, das gleichzeitig sowohl Nutzdaten (*Stamm- und Bewegungsdaten*) als auch die Konfigurationseinstellungen (*Customizing-Daten*) verwaltet. Sie sind aufgrund ihrer datenbankzentrierten Architektur meist monolithisch, da als Integrationsbasis vornehmlich die Datenbank und nicht funktionale Kompositionsmechanismen gewählt werden. Teilfunktionalitäten dieser Systeme können

deshalb auch schwer ersetzt werden. Hier erhofft man sich von der komponentenbasierten Anwendungsentwicklung (Ansatz 5) wesentliche Fortschritte.

Solche komponentenbasierten Systeme werden über Schnittstellen, die standardisierte Dienste anbieten, miteinander verbunden. Das hat u.a. den Vorteil, dass Teilkomponenten, die nicht den Anforderungen des Anwenders entsprechen, leicht ausgetauscht werden können. Dadurch können diese Systeme – vorausgesetzt, es etabliert sich tatsächlich in Zukunft ein hinreichend großer Softwaremarkt für solche Komponenten – enger an den Bedürfnissen der Anwender ausgerichtet werden. Die generative Softwareentwicklung (Ansatz 3) hat bisher nur in Nischenbereichen zu Erfolgen geführt. Gerade im Bereich der betrieblichen Anwendungssysteme sind jedoch noch eine große Zahl von Problemen vorhanden, die teilweise noch nicht einmal für die einfacher zu bewerkstelligende komponentenbasierte Softwareentwicklung gelöst sind. Dazu gehören Fragen zu den Themenbereichen Schema- und Schnittstellenevolution (siehe z.B. [Tan und Katayama 1989]), die bei generierten Systemen automatisch berücksichtigt werden müssten, jedoch bei komponentenbasierten Systemen ausschließlich im Verantwortungsbereich menschlicher Akteure, nämlich der Hersteller von Komponenten und Anwendungssystemen liegen.

1.2 Ein Gestaltungsprozess für komponentenbasierte Anwendungen

In dieser Arbeit wird davon ausgegangen, dass sich der Gestaltungsprozess komponentenbasierter Anwendungssysteme (siehe Abbildung 1.1 und [Sandmann 1999]) als übergeordneter Entwicklungsprozess (*Makroprozess*) grob in vier Entwicklungsprozesse (*Mikroprozesse*) untergliedern lässt und zwar in

- die *Entwicklung* atomarer, möglichst domänenunabhängiger Komponenten,
- die *Entwicklung* komplexer, domänenspezifischer Komponenten – eine vollständige, d.h. eigenständig einsetzbare Komponente heißt auch *Anwendungssoftwareprodukt* –,
- die *Konfiguration* von Anwendungssoftwareprodukten zu *Anwendungssystemen* und
- die *Anwendung*/Nutzung eines Anwendungssystems.

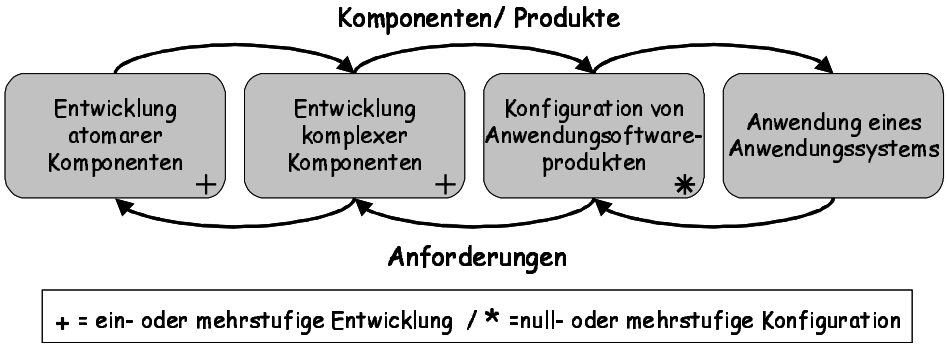


Abbildung 1.1: Makroprozess komponentenbasierter Anwendungsentwicklung

Die atomaren Komponenten kann man weiterhin in *geschlossene* und *parametrierbare* Komponenten, sogenannte *Komponenten-Frameworks*, unterteilen. Auf Basis solcher Frameworks können dann komplexe Fachkomponenten durch Einsetzen anderer, evtl. ebenfalls komplexer Komponenten konstruiert werden. Dazu werden die einfacheren Komponenten in dafür vorgesehene Steckplätze, sogenannte *Slots*, der komplexen Komponenten eingesetzt. Eine Beispielstruktur ist in Abbildung 1.2 zu sehen.

Bei der Entwicklung der atomaren Komponenten kann noch differenzierter in System- und Fachkomponenten – sowohl in Form geschlossener Komponenten als auch in Form von Frameworks – unterschieden werden. So wird in [Szyperski 1998] die erste Aufgabe (Abbildung 1.1) in die Teilaufgaben „Entwicklung von Komponentensystemen“ (sogenannte *System-Frameworks*), „Entwicklung von Komponenten-Frameworks“ und „Entwicklung von (komplexen) Komponenten“ unterteilt. System-Frameworks dienen der Integration verschiedener fachlicher Komponenten-Frameworks, die wiederum andere Komponenten beherbergen. Dieser durchaus nachvollziehbaren, aber dennoch bis heute in der Praxis nicht vollständig umgesetzten Unterscheidung wird nicht gefolgt, da sie für die anwendungsbezogene Betrachtungsweise dieser Arbeit nicht von Bedeutung ist.

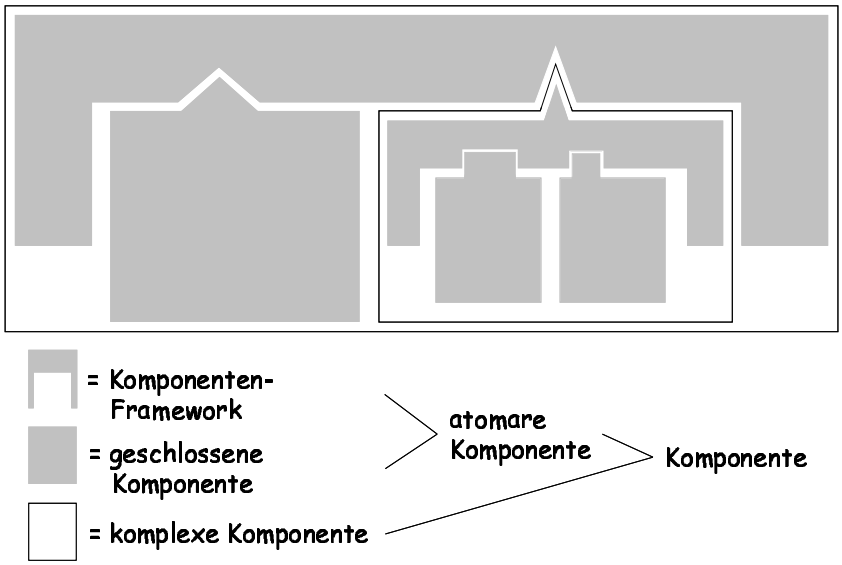


Abbildung 1.2: Framework-basiertes Anwendungssystem

In dieser Arbeit werden Komponenten nur in geschlossene Komponenten und Komponenten-Frameworks, die zum Aufbau komplexer Fachkomponenten genutzt werden, unterteilt. Dieses Konstruktionsprinzip lässt sich rekursiv wiederholen. Analog zu dem rekursiven Konstruktionsprinzip lässt sich auch der Gestaltungsprozess iterieren – dieser Sachverhalt wird in Abbildung 1.1 durch das „+“-Zeichen symbolisiert –, so dass Fachkomponenten nach und nach durch unterschiedliche Hersteller zu größeren Komponenten (bis zu Anwendungssoftwareprodukten) aufgebaut werden können.

Die *Konfiguration* wird in den Arbeiten zum Thema „komponentenbasierter Softwareentwicklung“ wie z.B. [Szyperski 1998] und [Griffel 1998] kaum berücksichtigt und wenn, dann nur aus einer eher technischen Software-Engineering-Perspektive. In dem hier dargestellten Gestaltungsprozess wird davon ausgegangen, dass ein Anwendungssoftwareprodukt i.d.R. nicht sofort einsetzbar ist, sondern noch konfiguriert werden muss. Diese Konfiguration bezieht sich nicht nur auf die Anpassung einzelner Komponenten (*Parametrierung*), sondern insbesondere auch auf die Auswahl von alternativen Komponenten, die zur Erweiterung von Frameworks zur Verfügung stehen. So kann beispielsweise ein Komponenten-Framework für den Bereich der Produktionsplanung eine Erweiterungsmöglichkeit (Slot) für Algorithmen zur Feinterminierung besitzen, für die drei Alternativen, z.B. auf Basis genetischer, wissensbasierter oder constraint-basierter Verfahren, zur Auswahl angeboten werden. Der Anwender muss sich dann entscheiden, welchen Algorithmus er einsetzen möchte.

Diese Art der Konfiguration kann recht komplex werden, wenn man sich den Konfigurationsvorgang (Customizing) heutiger monolithischer Standardsoftwareprodukte vor Augen führt (siehe z.B. [Dittrich, Mertens und Hau 1999]). Durch die Komponentenbasierung wird der Softwaremarkt wesentlich vielfältiger und die dadurch rein rechnerisch denkbaren Konfigurationsmöglichkeiten noch unüberschaubarer. Idee der vorliegenden Arbeit ist es deswegen, durch die Vorbereitung sinnvoller, anwendungsbezogener Konfigurations-alternativen in Form von (konfigurierbaren) Anwendungssoftwareprodukten die Komplexität dieses Problems maßgeblich einzuschränken. So wird nicht der vielfach vertretenen These gefolgt, dass der Anwender selbständig die alternativen Komponenten auf dem Markt auswählt und sich daraus ein Anwendungssystem zusammensetzt. Statt dessen kann nach der hier vorliegenden Vision der Anwender bei entsprechenden Herstellern vollständige, komponentenbasierte Anwendungssoftwareprodukte – die natürlich immer noch individuell erweiterbar sind – lizenzieren. Diese Softwareprodukte sind meist noch nicht direkt einsetzbar, sondern müssen durch einen Konfigurierer zu einem konkreten Anwendungssystem konfiguriert werden.

Die *Anwendung* der entwickelten Systeme gehört mit in den Gestaltungsprozess, da in der iterativen betrieblichen Anwendungssystemgestaltung eine ständige Verbesserung und Erweiterung der Systeme angestrebt wird, die ohne die Partizipation des Anwenders undenkbar wäre.

Diese vier Phasen (Herstellung von atomaren und komplexen Komponenten sowie die Konfiguration und Anwendung von Anwendungssystemen) können u.U. mehrfach nacheinander ausgeführt werden. So können Hersteller von atomaren Basiskomponenten genauso wie Hersteller von komplexen Fachkomponenten untereinander Zulieferbeziehungen haben: So liefert dann z.B. ein Hersteller von Basiskomponenten seine Produkte an den nächsten Hersteller, der Basiskomponenten einer nächst höheren Aggregationsstufe

konstruiert. Gleiches gilt für die Hersteller komplexer Komponenten und die Konfigurierer. Für die Konfigurierer sieht der Sachverhalt allerdings etwas anders aus, da die „Add-Ons“ eines Konfigurierers nicht zusätzliche Komponenten, sondern zusätzliche Konfigurationsentscheidungen sind, die er an einem Anwendungssoftwareprodukt vornimmt. Denkbar sind dann z.B. *branchentypische Vorkonfigurationen* (*pre-customizing*), wie sie beispielsweise Beratungsfirmen im R/3-Umfeld heute bereitstellen, um den Konfigurationsaufwand zu reduzieren. Sie werden dann durch einen Konfigurierer beim Anwender zu konkreten Anwendungssystemen vervollständigt.

Die vier Phasen, mit denen jeweils eigene komplexe Entwicklungsprozesse verbunden sind, können im Vergleich zu dem übergeordneten Gestaltungsprozess (Makroprozess) als Mikroprozesse bezeichnet werden. Sie werden i.d.R. durch verschiedene Akteure (Softwareentwickler, Unternehmen, Abteilungen, Mitarbeiter o.ä.) koordiniert, für die nun abstrakte Rollen definiert werden. Diese Rollen werden analog zu den Aufgaben als *Komponentenhersteller*, *Anwendungsarchitekt*, *Konfigurierer* und *Anwender* bezeichnet. Welche konkreten Akteure die einzelnen Rollen besetzen, kann von Fall zu Fall unterschiedlich sein. Dabei können durchaus komplexe Akteure mehrere Rollen gleichzeitig innehaben. Beispielsweise werden große Unternehmen Komponenten von externen Komponentenherstellern beziehen, aber zusätzlich auch spezielle Komponenten in Eigenregie erstellen, um letztlich komplexe Komponenten konstruieren zu können, die ihren speziellen Anforderungen gerecht werden. Die so entstehenden Anwendungssoftwareprodukte können dann intern für verschiedene Werke individuell konfiguriert und eingesetzt werden.

Kleine Unternehmen legen vielfach mehr Wert auf Komplettlösungen. Sie können deshalb von Beratungs- oder Softwareunternehmen vollständig konfigurierte Lösungen beziehen.

1.3 Modelle/Beschreibungen von Komponenten und Anwendungssystemen

Zur Entwicklung, Auswahl und Konfiguration betrieblicher Anwendungssysteme werden entsprechende Methoden und Werkzeuge benötigt, die u.a. der Beschreibung, dem Finden und dem Vergleich von Komponenten dienen. Sie sollen das Arbeiten mit den Komponenten vereinfachen, indem sie möglichst von allen unwichtigen, technischen Informationen abstrahieren und die für den Anwender wichtigen fachlichen Gestaltungspotentiale der Komponenten in den Vordergrund stellen. In dieser Arbeit werden Sprachen entwickelt, mit denen Beschreibungen für einfache (atomare) Komponenten bis hin zu komplexen Anwendungssoftwareprodukten erstellt werden können. Diese Beschreibungen sollen die Komponenten für die am Entwicklungsprozess beteiligten Akteure verständlicher machen und die Akteure in ihrer Aufgabenerfüllung unterstützen.

Heute dienen schon Softwarereferenzmodelle wie das R/3-Referenzmodell der SAP (siehe [Keller und Teufel 1997] und Abschnitt 4.2.3) der Unterstützung des Konfigurationsprozesses (Customizing). Softwarereferenzmodelle beschreiben dazu die Software aus Sicht der mit ihr

realisierbaren Geschäftsprozesse. Diese Prozesse dienen der anwendungsbezogenen Beschreibung des Systems und stellen neuerdings (siehe [Appelrath und Ritter 1999]) auch die Auswahlkriterien dar, anhand derer der *Konfigurationsraum* – also die Menge der Konfigurationsalternativen – im Rahmen der Einführung eingeschränkt werden kann. Jedoch haben die dabei verwendeten Sprachen – im Fall des R/3-Referenzmodells werden zur Beschreibung der Geschäftsprozesse Ereignisgesteuerte Prozessketten (EPK, siehe Abschnitt 4.2.2 und [Scheer 1998a]) eingesetzt – zahlreiche Schwächen. So ist beispielsweise die EPK weder syntaktisch vollständig und formal definiert, noch ist ihre Semantik eindeutig. Verschiedene Versuche, ihre Syntax und Semantik (siehe [Chen und Scheer 1994], [Langner, Schneider und Wehle 1997], [Weikum et al. 1997] und [Rump 1999]) zu definieren, beschränken sich ausschließlich auf formale Semantiken (z.B. durch Übersetzung in Petri-Netze oder Transitionssysteme), von denen der Anwender jedoch wenig Nutzen hat, denn gerade diese Modelle sollen dem Menschen als Interpretationshilfe des Anwendungssoftwareprodukts dienen und nicht in erster Linie maschineninterpretierbar sein.

Eine Beschreibungssprache für Komponenten muss deswegen neben einem formalen auch ein inhaltliches Fundament besitzen, das sich an der natürlichen Sprache orientiert. Die natürliche Sprache ist die einzige Sprache, die jeder Mensch gesprochen und/oder geschrieben versteht und die kaum zusätzlicher Interpretationsunterstützung bedarf.

Aber warum kann eine Sprache nicht vollständig neu erfunden werden, und warum muss statt dessen auf die zweifellos problembehaftete, natürliche Sprache zurückgegriffen werden? Das Problem ist und bleibt das des methodischen Neuanfangs. Alles was man neu definiert, basiert auf der natürlichen Sprache. Wie in dem Vergleich von Neurath (siehe [Neurath 1979]) treffend geschildert wurde, kann man sich die Definition einer neuen Umgangssprache wie den Versuch vorstellen, ein neues Schiff zu bauen, während man sich mit einem havarierten Schiff auf offener See befindet. Da kein Trockendock und keine neuen Materialien zur Verfügung stehen, kann man nur auf Bestehendes, z.B. herumtreibende Holzteile – im übertragenen Sinn: Begriffe des bestehenden Sprachumfangs –, zurückgreifen.

Somit kann keine Sprache ohne Rückgriff auf die natürliche Sprache definiert werden. Und ohne die natürliche Sprache als Mittel zur symbolischen Referenz und Abstraktion würde alles auf der Welt individuell und konkret bleiben. Die natürliche Sprache hat sich durch einen fortlaufenden Abstraktionsprozess nach und nach zu dem entwickelt, was sie heute ist: das Fundament der Wissenschaften und der gesamten menschlichen Zivilisation.

Ein Problem bleibt jedoch bestehen: die Semantik der natürlichen Sprache ist nicht eindeutig. Sie hat einige „Defekte“ wie Homonyme, Synonyme, Mehrdeutigkeiten von Sätze usw., die es zu beheben gilt, damit die durch sie geschilderten Sachverhalte eindeutig interpretierbar werden. Aus diesem Grund wird vermehrt auf „fast“ natürliche Sprachen (*Normsprachen*, *Orthosprachen*) zurückgegriffen, wie z.B. in den Informatikarbeiten von [Wedekind 1992], [Ortner 1997] und [Schienmann 1997].

1.4 Hypothesen der Arbeit

Das Forschungsgebiet komponentenbasierter Softwareentwicklung i. Allg. und der fachlich ausgerichteten Komposition von betrieblichen Anwendungssystemen aus standardisierten, austauschbaren Komponenten im Speziellen ist noch sehr jung. Deshalb kann über die Frage, „wohin die Reise geht“, größtenteils nur spekuliert werden. Folglich müssen in dieser Arbeit eine Reihe von Annahmen getroffen werden, für die nur einige – sicherlich nicht ausreichende – Argumente gebracht werden können, die aber im weiteren Verlauf der Arbeit vorausgesetzt werden sollen. Sie entspringen teilweise den persönlichen im universitären und auch im betrieblichen Umfeld bei zwei Arbeitgebern erworbenen Erfahrungen des Autors, die aber bisher nicht systematisch untersucht werden konnten und deswegen nur als Hypothesen verstanden werden dürfen:

Hypothese 1: Es wird sich früher oder später ein Softwaremarkt für domänenspezifische Komponenten (Fachkomponenten) herausbilden.

> Diese Hypothese kann durch Extrapolation bestehender Trends unterstützt werden. Im Bereich der Benutzungsschnittstellenentwicklung (GUI-Entwicklung) hat sich bereits ein solcher Markt etabliert. Komponenten auf Basis von OLE- oder JavaBeans-Technologie sind dafür eine Bestätigung. Außerdem haben schon eine Reihe großer Softwareunternehmen damit begonnen, ihre Produkte serverseitig in fachliche Komponenten zu zerlegen.

Hypothese 2: Die Qualität und Flexibilität betrieblicher Anwendungssysteme wird sich durch die komponentenbasierte Anwendungsentwicklung erhöhen.

> Monolithische Anwendungssysteme wie R/3 oder BaanERP werden schon heute von ihren Herstellern gegenüber spezialisierten Softwarelösungen anderer Hersteller geöffnet, wodurch sich u.a. die Flexibilität dieser Systeme erhöht – der erste Schritt in eine Komponentenwelt. Die Qualität wird durch die Möglichkeit der einfacheren Austauschbarkeit und den dadurch wiederum stärker werdenden Konkurrenzdruck zwangsläufig steigen.

Hypothese 3: Die Komplexität der rein komponentenbasierten Anwendungssystemgestaltung wird den Anwender (= das anwendende Unternehmen) in Zukunft überfordern. Aus diesem Grund werden vorgefertigte Kompositionsmöglichkeiten benötigt, die den Aufwand der Suche nach geeigneten Komponenten und der anschließenden Komposition und/oder Konfiguration stark reduzieren.

> Schon heute, im Zeitalter hochgradig konfigurierbarer, monolithischer Anwendungssoftwareprodukte kommt ein Anwender i.d.R. nicht ohne externe Dienstleister (z.B. Anwendungsberater) aus. Die Variantenvielfalt betrieblicher

Anwendungssysteme wird durch die Komponentenorientierung um ein Vielfaches erhöht. Der Anwender wird – anstatt selbst die Komponenten zu vollständigen Anwendungssystemen zusammenzubauen – auf Spezialisten (Anwendungsarchitekten und Konfigurierer) zurückgreifen, die aus Komponenten spezielle Anwendungssoftwareprodukte zusammensetzen und betriebliche Anwendungssysteme konfigurieren.

Hypothese 4: Die vorherrschende Sichtweise des Anwenders betrieblicher Softwaresysteme ist geschäftsprozessorientiert.

Die Potentiale der Gestaltung betrieblicher Anwendungssysteme liegen zunehmend in der Gestaltung des Verhaltens dieser Systeme, also von Reihenfolge (Koordination) sowie Art und Weise der Manipulation betrieblicher Geschäftsobjekte, weniger aber in der Gestaltung der Struktur dieser Objekte, die mehr und mehr zugunsten der zwischen- und innerbetrieblichen Kommunikation standardisiert werden.

Hypothese 5: Da die natürliche Sprache der Ausgangspunkt aller künstlichen Sprachen ist, muss sie aus Sicht des Anwenders konsequent als Grundlage zur Beschreibung und zur Interpretation von Softwarekomponenten genutzt werden.

Softwarekomponenten sind formale Konstrukte, deren Bedeutung erst in einem Anwendungsbereich durch Interpretation zur Geltung kommt. Für den Anwender stellt die natürliche Sprache die Grundlage für das Verstehen einer Softwarekomponente dar. Deswegen sollten alle Beschreibungen systematisch auf die natürliche Sprache zurückgeführt werden. Allerdings muss die natürliche Sprache dazu um einige „Defekte“ bereinigt werden.

1.5 Zielsetzung der Arbeit

In dieser Arbeit sollen Sprachen entwickelt werden, die der anwendungsbezogenen Beschreibung von Komponenten und daraus aufgebauter Anwendungssysteme dienen. Diese Sprachen bzw. die mit ihnen formulierten Beschreibungen sollen die Aufgaben der komponentenbasierten Konstruktion und Konfiguration betrieblicher Anwendungssysteme vereinfachen.

Eine der zu entwickelnden Sprachen soll zur Beschreibung verhaltensbezogener Aspekte von Komponenten dienen. Dabei liegt das Hauptaugenmerk auf dem aufrufbaren Verhalten einer Komponente und nicht etwa auf dem beobachtbaren Verhalten, das bei der Untersuchung von Sicherheitsanforderungen relevant wäre. Grundsätzlich sollen die Komponenten-beschreibungen von den für den Anwender unwichtigen Implementierungsdetails abstrahieren und systematisch zu größeren Beschreibungen zusammengesetzt werden können – genauso, wie die durch sie beschriebenen Komponenten zu großen Systemen zusammengesetzt werden

können. Sie sollen den verschiedenen Akteuren (Anwendungsarchitekten, Konfigurierern, Anwendern), die am Entwicklungsprozess komponentenbasierter Anwendungssysteme (Makroprozess) beteiligt sind, als Grundlage für ihre Tätigkeiten (Zusammenbau und Konfiguration der Komponenten) dienen.

Neben den Komponenten sollen auch ausgewählte, für bestimmte Anwendungsbereiche besonders geeignete Zusammenstellungen (Komposita) der Komponenten – eine Menge solcher Komposita heißt dann *Anwendungssoftwareprodukt* – beschrieben werden. Dazu soll eine zusätzliche Konfigurationssprache vorgeschlagen werden, in der sowohl die Komponenten – nun aus Sicht der Konfiguration – als auch die für einen betrachteten Anwendungsbereich „sinnvollen" Komposita beschrieben werden. Ein einzelnes Kompositum beschreibt letztlich ein konkretes *Anwendungssystem*.

Das Problem der Substituierbarkeit von Komponenten sowie (halb-)automatische Verfahren zum Beweisen, dass eine Komponente zu einer anderen vorgegebenen Komponente „passt", werden in dieser Arbeit nur am Rande behandelt.

Zur Bewertung der Komposita sollen die Konsequenzen, die sich aus einem möglichen Einsatz der ausgewählten Komponenten ergeben, anhand zukünftiger Geschäftsprozesse[1], die mittels dieser Komponenten abgewickelt werden können, verdeutlicht werden. Dazu wird in dieser Arbeit gezeigt, wie sich die Geschäftsprozessbeschreibungen systematisch aus den Komponentenbeschreibungen ableiten lassen. Damit dazu nicht erst ein Kompositum vollständig erstellt werden muss, sollen andersherum den Anforderungen entsprechende Komposita durch die Ableitung von Geschäftsprozessen („*Prozessorientierte Konfiguration*") ermittelt werden. Dabei kann die Tatsache berücksichtigt werden, dass aus der Menge alternativer Komponenten erst dann eine Komponente bzw. ihre Beschreibung ausgewählt werden muss, wenn sie im Rahmen einer Geschäftsprozessbeschreibung auch tatsächlich benötigt wird. Die Auswahl von Komponenten hat wiederum Auswirkungen auf die Auswahl anderer Komponenten, die automatisch durch ein Konfigurationssystem propagiert werden müssen und somit Einfluss auf die Gestaltung der weiteren Geschäftsprozesse nehmen. Dieser Sachverhalt wird in Abbildung 1.3 grafisch veranschaulicht. Im Rahmen einer Konfiguration werden dort Geschäftsprozessbeschreibungen abgeleitet, die z.B. mit der Eingabe eines Fertigungsauftrages beginnen und nach der Disposition benötigter Materialien mit einer Nachkalkulation schließen. Ein Anwendungssoftwareprodukt für solche Geschäftsprozesse besteht im Beispiel aus drei Komponenten: einem Produktionsplanungssystem (PPS), einer

[1] Es handelt sich dabei – wie im R/3-Referenzmodell der SAP – nur um Geschäftsprozesse, deren Interaktionen unmittelbar im Zusammenhang mit dem jeweiligen Anwendungssystem stehen. In UML (siehe [Booch, Rumbaugh und Jacobson 1997a]) wird analog von Use Cases gesprochen, die durch Interaktions- oder Kollaborationsdiagramme spezifiziert werden.

Materialwirtschaft (MAWI) und einer Finanzbuchhaltung (FIBU). Für jede dieser Komponenten stehen zwei bzw. drei konkrete Komponenten zur Auswahl zur Verfügung. Eine Beschreibung eines Geschäftsprozesses kann nun beispielsweise durch eine Interaktion „Eingabe eines Fertigungsauftrags“ mit dem System begonnen werden. Je nach ausgewähltem Auftragstyp (z.B. Einzel- oder Massenfertigung) wird dadurch eine entsprechende Komponente – beispielsweise eine Einzelfertigungs- oder eine Massenfertigungskomponente – durch den Konfigurierer bestimmt. Von dieser Auswahl kann unter Umständen auch die Wahl der benötigten MAWI-Komponente abhängen. Für Einzelfertigungsaufträge werden beispielsweise die benötigten Materialien meist bedarfsgesteuert disponiert, während die Materialbeschaffung im Rahmen der Massenfertigung verbrauchsgesteuert abläuft. Somit kann ein Konfigurationssystem durch Propagierungen entsprechender Abhängigkeiten selbsttätig dafür sorgen, dass im Rahmen der Beschreibung eines Geschäftsprozesses die richtigen Komponenten und damit Interaktionen im Geschäftsprozess durchgeführt werden. Analog werden entsprechende Komponenten für die Kalkulation für das interne Rechnungswesen (FIBU) ausgewählt. Dies kann wieder entweder manuell durch den Konfigurierer oder aufgrund entsprechender Regeln automatisch geschehen.

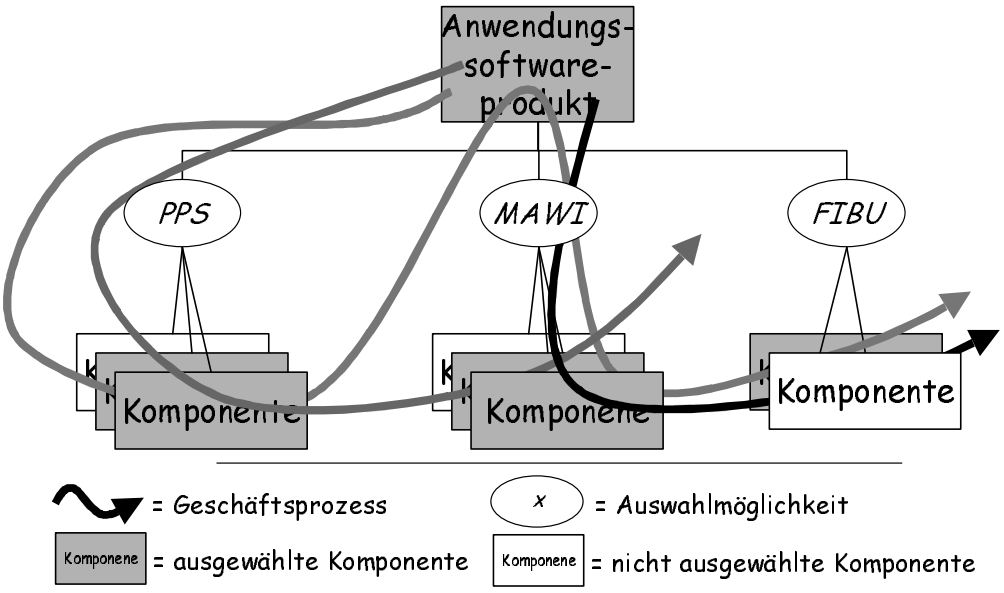


Abbildung 1.3: Prozessorientierte Konfiguration

Neben diesen bisher eher formalen Aspekten sollen in dieser Arbeit Methoden entwickelt werden, die es Anwendungsarchitekten und Konfigurierern ermöglichen, im Rahmen des Makroprozesses für den Anwender inhaltliche Interpretationen in die Beschreibungen der Komponenten bzw. Anwendungssysteme einzubringen. Aus formal-logischer Sicht werden unter Interpretationen (synonym: Strukturen) Abbildungen verstanden, die den Symbolen (Konstanten und Logiksymbole) und Ausdrücken (Prädikaten) einer formalen Beschreibung Mengen von Extensionen einer „möglichen Welt“ zuordnen. Für den Anwender ist jedoch

eine i.d.R. aufwendig zu formulierende Zuordnung der Extensionen kaum zumutbar. Einfacher ist es, die Terminologie des Anwenders als Interpretation in die Komponentenbeschreibungen einzubringen und zu prüfen, welche Konsequenzen sich daraus ergeben.

Wenn z.B. in einer Komponentenbeschreibung das Symbol „Kundenauftrag“ benutzt wird, jedoch der Anwendungsarchitekt in seiner Unternehmenswelt statt dessen von „Bestellungen“ sprechen möchte, so kann er (testweise) das Symbol „Kundenauftrag“ durch das Symbol „Bestellung“ ersetzen. Ob die „Bestellung“ tatsächlich der (intensionalen) Semantik von „Kundenauftrag“ entspricht, kann er dann anhand der Konsequenzen überprüfen, die sich aus dieser Ersetzung ergeben, da sie sich durch alle Komponenten- und folglich auch durch die abgeleiteten Geschäftsprozessbeschreibungen zieht. Neben einfachen Symbolen können auch Satzformen ersetzt werden, so dass auch komplexere Zusammenhänge entsprechend interpretiert werden können. Die Satzformen könnten z.B. der natürlichen Sprache entliehen werden. Da die natürliche Sprache jedoch „Defekte“ (z.B. Mehrdeutigkeiten) aufweist, wird eine Normsprache vorgeschlagen, die gerade dieses Problem umgeht.

Prinzipiell wäre es möglich die Komponenten und daraus erstellte Anwendungssysteme sowie die an sie gestellten Anforderungen vollständig formal zu beschreiben. Andererseits haben solche formalen Beschreibungen heute in der betrieblichen Praxis keine Bedeutung, da sie für den Anwender, aber auch für die anderen am Entwicklungsprozess beteiligten Akteure wie Konfigurierer und Anwendungsarchitekten, zu komplex und unverständlich sind. Die in dieser Arbeit entwickelten Sprachen sollen grundsätzlich einen formaleren, teilweise automatisierten Abgleich von Komponenten und Anwendungssystemen mit den Anforderungen des Anwenders erlauben, als er letztlich hier entwickelt wird. Diese formalen Ansätze werden wegen der mangelnden Praxistauglichkeit nicht weiter verfolgt. Statt dessen wird ein Ansatz angestrebt, der den Anwendungsarchitekten, Konfigurierern und Anwendern eine inhaltliche Interpretation erlaubt. Diese Art der Interpretation des „formalen“ Anwendungssystems, soll eine Veranschaulichung der Konsequenzen, die sich aus dem Einsatz eines Anwendungssystems ergeben, in einer für die Akteure „verständlichen“ Sprache erlauben.

1.6 Gliederung der Arbeit

Die Arbeit gliedert sich in fünf Teile (siehe zum Überblick auch Abbildung 1.4).

In Teil I werden die wichtigsten fachlichen Grundlagen erarbeitet. Dabei spielen die Aspekte „Betriebliche Anwendungssysteme“ (Kapitel 2), „Komponentenbasierte Anwendungssysteme“ (Kapitel 3) und „Sprachen zur Beschreibung von Komponenten“ (Kapitel 4) eine wesentliche Rolle. In Kapitel 5 wird die erwähnte Normsprache näher vorgestellt.

In Teil II wird das grundlegende Konzept der „prozessorientierten Konfiguration“ dieser Arbeit hergeleitet. Dazu wird zunächst die Problemstellung vertieft (Kapitel 6); anschließend wird ein formales Komponentenmodell konstruiert (Kapitel 7), das auf einer Beschreibungssprache für komponentenbasierte Anwendungssoftwareprodukte basiert. In Kapitel 8 werden die Zusammenhänge zwischen Komponentenbeschreibungen und ihren normsprachlichen Interpretationen genauer erarbeitet. Die Konfigurationssprache zur Beschreibung von konfigurierbaren Anwendungssoftwareprodukten wird in Kapitel 9 vorgestellt. Kapitel 10 definiert die prozessorientierte Konfiguration unter Berücksichtigung der zuvor erarbeiteten Konzepte vollständig.

In Teil III wird die Implementierung des PROSECCO genannten Gesamtsystems vorgestellt.

In Teil IV wird die Arbeit zusammengefasst, eine Kritik der vorgestellten Arbeit vorgenommen sowie ein Ausblick auf mögliche Anschlussarbeiten gegeben.

Teil V bietet Anhänge in denen weitere Grundlagen kurz aufbereitet werden. U.a. werden die formalen Grundlagen des π-Kalküls in Kapitel 17 und von Beschreibungslogiken in Kapitel 18 vorgestellt.

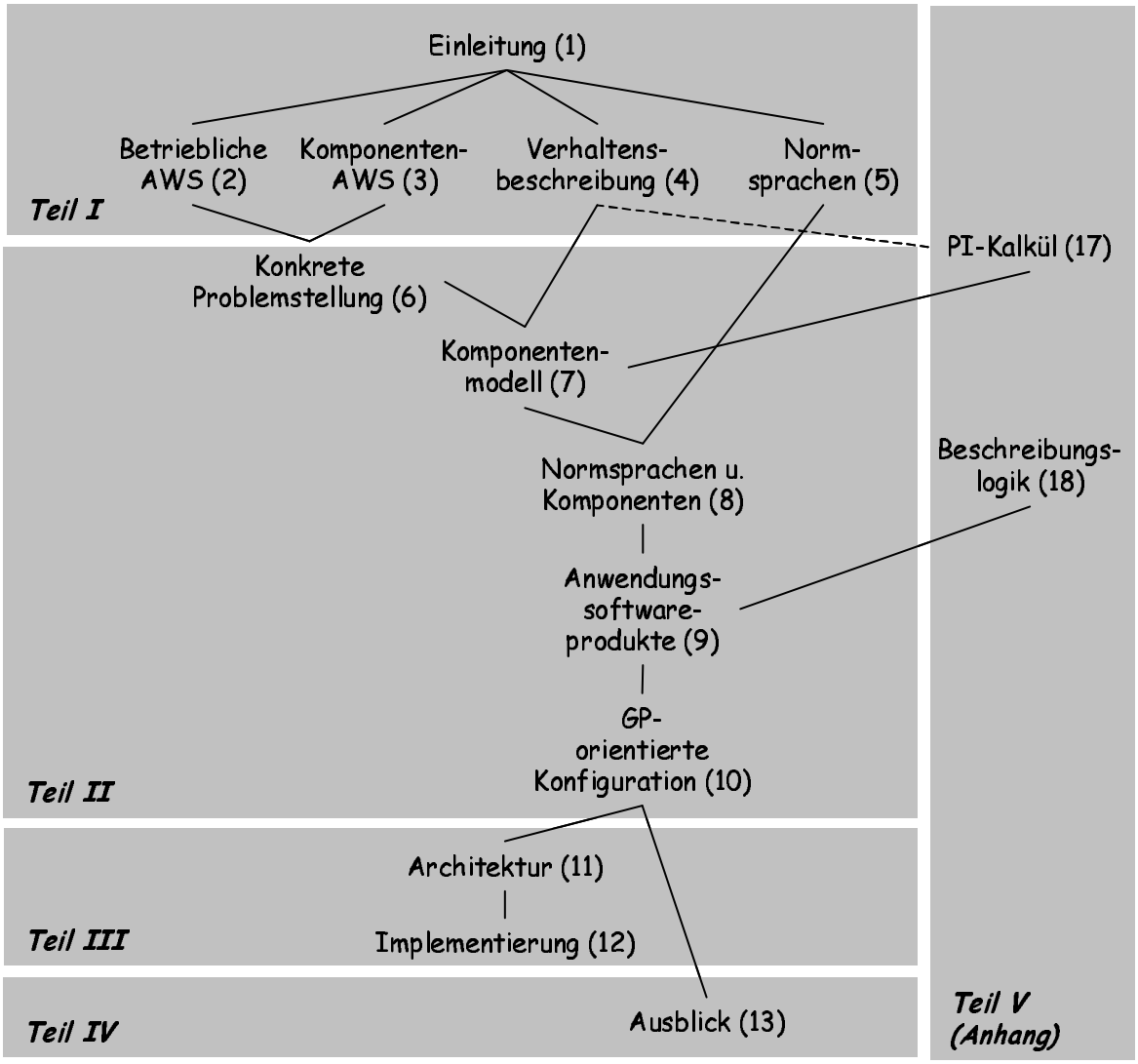


Abbildung 1.4: Die Kapitel der Arbeit und ihre Abhängigkeiten

2 Betriebliche Anwendungssysteme

Betriebliche Anwendungssysteme und ihre Anpassung an die Anforderungen des anwendenden Unternehmens stellen den Einsatzbereich der in dieser Arbeit erarbeiteten Methoden und Werkzeuge dar. Im Abschnitt 2.1 werden zunächst die Grundlagen dieser Systeme näher betrachtet. Der Ausgangspunkt ist das „Gesamtsystem Unternehmen“, das durch seine Ziele die Anforderungen an seine Teilsysteme und damit insbesondere an das Teilsystem „betriebliches Anwendungssystem“ stellt. Zur Realisierung von Anwendungssystemen werden heute vornehmlich monolithische Standardsoftwareprodukte eingesetzt, deren Konfigurationsmöglichkeiten (Abschnitt 2.2) die Berücksichtigung unternehmensindividueller Anforderungen ermöglichen. Das Ziel dieser Konfiguration ist eine adäquate Umsetzung der fachlichen Anforderungen durch technische Justierung des Softwareprodukts. Zur Überführung der fachlichen Anforderungen in die technischen Einstellungen gibt es verschiedene Konfigurationshilfsmittel, die in Abschnitt 2.3 vorgestellt werden.

2.1 Grundlagen

2.1.1 Teilsysteme des Unternehmens

Im Sinne einer Top-down-Betrachtungsweise ist das Unternehmen das Gesamtsystem, in das das betriebliche Anwendungssystem als Teilsystem eingeordnet ist. Das Unternehmen definiert die Randbedingungen und Anforderungen aus denen sich ein betriebliches Anwendungssystem ableiten lässt. Unabhängig davon, welche Art von Software entwickelt und eingeführt werden soll, müssen diese Bedingungen berücksichtigt werden. Aus diesem Grund wird das Unternehmen und seine Teilsysteme im Folgenden näher betrachtet.

Ein Unternehmen (siehe Abbildung 2.1 und [Ferstl und Sinz 1998]) kann gemäß der Typen seiner zu verarbeitenden Objekte in die Teilsysteme (betriebliches) Informationssystem und Basissystem gegliedert werden. Im Informationssystem sind Objekte vom Typ Information und im Basissystem, das hier nicht weiter betrachtet werden soll, alle anderen, meist physischen Gegenstände wie Maschinen und Materialien zu finden. Unter einem *Informationssystem* soll also das gesamte informationsverarbeitende Teilsystem eines Unternehmens verstanden werden. Dieses Teilsystem wird nicht nur durch technische Aspekte, wie z.B. die verwendeten Rechner und Telekommunikationsanlagen, sondern auch durch soziale Aspekte, wie z.B. zwischenmenschliche Kommunikation und Formen der Teamarbeit bestimmt.

Betriebliche Anwendungssysteme gehören zu dem technischen Teil von Informationssystemen und sind gleichzeitig zu den automatisierten Aufgabenträgern eines Unternehmens zu zählen. Analog zu den Aufgaben eines Unternehmens kann man Anwendungssysteme inhaltlich

danach unterscheiden, ob sie mehr dem betrieblichen Lenkungssystem (Planung, Steuerung und Kontrolle) oder eher dem betrieblichen Leistungssystem (Administration, Disposition und Durchführung) zuzuordnen sind.

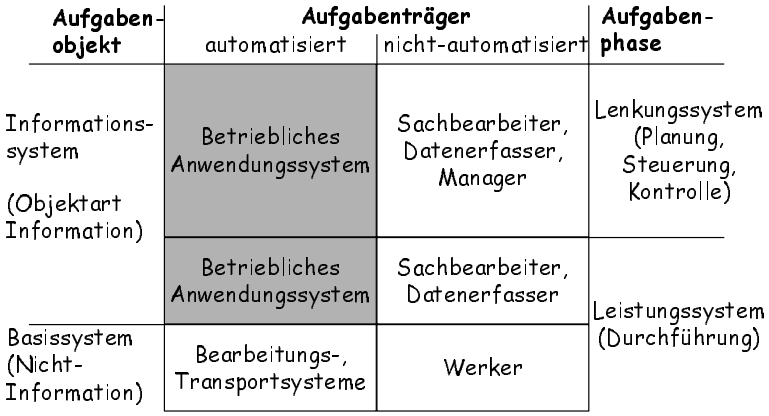


Abbildung 2.1: Teilsysteme des Unternehmens (siehe [Ferstl und Sinz 1998])

In einigen Zusammenhängen ist es sinnvoll, bei einem betrieblichen Anwendungssystem zwischen dem eigentlichen Softwareprodukt und einer Installation (inklusive Konfiguration) dieses Produkts zu unterscheiden. Ein *Anwendungssoftwareprodukt* wie das R/3-System der SAP, das als Produkt auf CD-ROM ausgeliefert wird, kann in mehreren Installationen (z.B. in unterschiedlichen Unternehmen oder Unternehmensbereichen) jeweils als Anwendungssystem dienen. In Zukunft wird also unter einem Anwendungssystem immer eine konkrete Installation eines Anwendungssoftwareprodukts auf einem Rechnersystem verstanden. Von den Anwendungssoftwareprodukten werden noch Systemsoftwareprodukte wie Betriebssysteme und Datenbanksysteme unterschieden, die zum Betrieb eines Anwendungssystems benötigt werden.

2.1.2 Unternehmensarchitektur

Betriebliche Anwendungssysteme stellen heute einen zentralen und strategisch bedeutsamen Bestandteil von Unternehmen dar. Sie müssen wie alle anderen Teilsysteme an den Zielen des Unternehmens ausgerichtet und systematisch aufgebaut werden. Aus der Sicht der für die Gestaltung betrieblicher Anwendungssysteme verantwortlichen Abteilung kann ein Unternehmen als dreischichtige Pyramide (siehe Abbildung 2.2) verstanden werden, deren Fundament (Ressourcen-Ebene) die Mitarbeiter und die betrieblichen Anwendungssysteme sowie Maschinen und Anlagen bilden. Mitarbeiter werden wegen der besseren Übersichtlichkeit und zur Komplexitätsreduzierung in Organisationseinheiten und Stellen einer *Aufbauorganisation* eingegliedert.

Mitarbeiter, Maschinen und betriebliche Anwendungssysteme agieren miteinander (interagieren). Abfolgen von Interaktionen zwischen ihnen, die der Verfolgung definierter Ziele dienen, werden als Geschäftsprozesse bezeichnet und in Form von *Geschäftsprozessmodellen* in der mittleren Schicht der Pyramide formuliert. Diese Modelle beschreiben die typischen Geschäftsabläufe *(Geschäftsprozesse)* eines Unternehmens, die durch Mitarbeiter und Anwendungssysteme unterstützt werden.

Die Ziele, die durch die Ausführung von Geschäftsprozessen erreicht werden sollen, und weitere Randbedingungen werden in der obersten Schicht in einem *Unternehmensplan* formuliert.

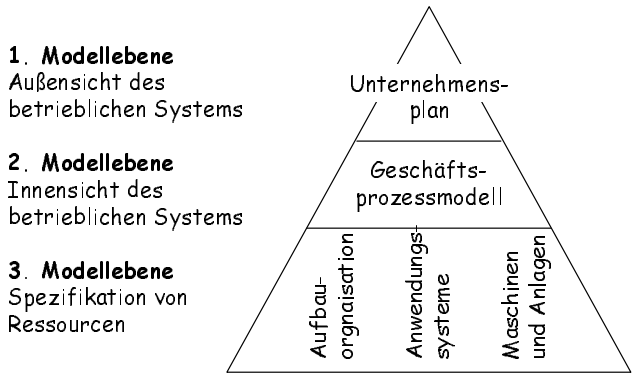


Abbildung 2.2: Unternehmensarchitektur (siehe [Ferstl und Sinz 1998])

Da am Anfang einer Unternehmensgründung ein System aus Zielen steht, das später ständig an die sich ändernden Rahmenbedingungen angepasst werden muss, versucht man idealerweise, Ablauf- und Aufbauorganisation sowie Ressourcen an diesen Zielen auszurichten (*Top-down-Vorgehensweise*) und nicht umgekehrt (*Bottom-up-Vorgehensweise*). Die tägliche Praxis zeigt aber, dass die Top-down-Vorgehensweise nicht immer praktikabel ist, denn oft müssen bestimmte Zustände im Unternehmen als gegeben hingenommen werden. So können Mitarbeiter nicht beliebig ausgetauscht oder versetzt, und alte Anwendungssysteme nicht „auf einen Schlag“ ersetzt werden, sondern müssen sukzessive abgelöst werden. Die wechselseitigen Beziehungen zwischen den angestrebten Unternehmenszielen bzw. den daraus abgeleiteten Geschäftsprozessmodellen (Soll-Zustand), dem augenblicklichen Zustand des Unternehmens (Ist-Zustand) und den Möglichkeiten betrieblicher Standardsoftwareprodukte machen die Auswahl und Einführung bzw. Entwicklung dieser Systeme zu einem komplexen Problem.

Neue Ziele ergeben sich oft durch das Aufdecken von Schwachstellen in der bestehenden Unternehmensorganisation oder durch Analysen der Umgebung (Markt, Konkurrenten) des Unternehmens.

Ein für diese Arbeit interessanter Aspekt dieser *Unternehmensarchitektur* ist, dass ein Unternehmen als parallele Komposition der drei „Komponenten“ Aufbauorganisation, Anwendungssystem und Anlagen (inkl. Maschinen) beschrieben werden kann, die der untersten Schicht in Abbildung 2.2 entspricht. Die Geschäftsprozessmodelle eines Unternehmens lassen sich dann konstruktiv – weil bottom up – aus den möglichen Kommunikationssequenzen dieser drei Teilkomponenten ableiten. Letztlich lassen sich aus den Geschäftsprozessmodellen die Ziele ermitteln, die mit den Geschäftsprozessen verfolgt werden sollen.

2.1.3 Monolithische Standardsoftware

Softwareprodukte können in systemnahe (z.B. Betriebssysteme, Datenbanksysteme) und anwendungsnahe Softwareprodukte klassifiziert werden. Letztere können weiter in Abhängigkeit des Grades ihrer Standardisierung in Individual- und Standardsoftware untergliedert werden. *Individualsoftware* wird für oder in einem Unternehmen selbst entwickelt und kann auf spezielle Einsatzbedürfnisse ausgerichtet werden. *Standardsoftware* wird dagegen möglichst flexibel und universell entwickelt, damit sie die Anforderungen möglichst vieler Unternehmen abdeckt. Standardisierung kann aufgrund von Gesetzen, Normen, wissenschaftlichen Erkenntnissen und durch die integriert angebotene Implementierung alternativer Verfahren erreicht werden. So haben Gesetze des Steuer- und Handelsrechts zu einer teilweisen Vereinheitlichung des externen Rechnungswesen geführt. Wissenschaftliche Erkenntnisse – z.B. im Bereich des internen Rechnungswesen mit den Verfahren der Kostenarten- und Kostenstellenrechnung – haben ebenfalls zur Vereinheitlichung der Methoden beigetragen. Unterstützt ein Softwareprodukt mehrere, alternative Verfahren, so ist es die Aufgabe des Anwenders (oder der durch ihn beauftragten Berater), das Produkt vor der Nutzung so einzustellen (zu konfigurieren), dass es seinen Anforderungen möglichst weitgehend gerecht wird. Da Standardsoftwareprodukte wie das R/3-System der SAP durch die zunehmende Integration und durch die Aufnahme alternativer Verfahren immer komplizierter werden, wird der Anwender vor eine oft unüberschaubare Variantenvielfalt gestellt, aus der er bei der Anpassung durch kundenspezifische *Parametrierung* (auch: Customizing) wählen muss. Die vor der Parametrierung notwendigen Schritte des Abgleichs der Anforderungen des Anwenders mit den Möglichkeiten der Software machen die Einführung standardisierter Software zu einem aufwendigen und dadurch teuren Prozess. Dies liegt u.a. daran, dass Unternehmen die Vor- und Nachteile der Verfahrensalternativen nicht beurteilen können bzw. gar nicht wissen, welche Strukturen und Prozesse des Unternehmens auf welche Art am besten abgebildet werden. So müssen auch mittelständische Unternehmen mit langen und kostspieligen Einführungsprojekten rechnen.

Trotz der weitgehenden Standardisierungsversuche einzelner Hersteller werden immer wieder individualisierte Lösungen benötigt, die in der Kombination von Standardsoftwareprodukten unterschiedlicher Hersteller wie Baan, Oracle und SAP liegen. Da die Produkte dieser

Hersteller heute oft vollständig mit ihrer gesamten Funktionalität installiert werden müssen, stellen in Zukunft kleinere Lösungen mit ausgewählten Komponenten, die nur die tatsächlich benötigte Teilfunktionalität eines Herstellers abdecken, sicherlich die elegantere Lösung dar. Dazu werden aber insbesondere herstellerübergreifende Standards benötigt, auf die sich die Hersteller einigen müssen.

Bei Standardsoftware können De-facto-Standards und die De-jure-Standards unterschieden werden. De-facto-Standards sind inoffizielle Standards, die auf den Entwicklungen eines Softwareunternehmens basieren und eine ausreichende Marktakzeptanz der Anwender erfahren, oder Richtlinien, die durch mehrere Unternehmen unterstützt werden. De-jure-Standards werden hingegen von offiziellen Organisationen wie z.B. der International Standards Organization (ISO) entwickelt. Sie gibt es bisher nur in Teilbereichen der betrieblichen Anwendungssysteme, z.B. im Bereich der Produktdatenmodelle mit ISO 10303, International Standard for the computer-interpretable representation and exchange of product data (STEP) und ISO 9735, Electronic data interchange for administration, commerce and transport, (EDIFACT).

Das R/3-System der SAP stellt z.B. einen De-facto-Standard für betriebliche Anwendungssysteme dar, der zum einen durch die große Marktakzeptanz bei den anwendenden Unternehmen und zum anderen durch die zunehmende Annäherung weiterer Softwareunternehmen (Complementary Software Partner) an die Vorgaben der SAP an Bedeutung gewinnt. Neben anderen De-facto-Standards – wie die MS-Office-Produkte – unterstützt die SAP natürlich auch gängige De-jure-Standards und ist an der Erarbeitung weiterer herstellerübergreifender Standards, z.B. zur Kopplung von Anwendungssystemen verschiedener Hersteller, beteiligt.

2.1.4 Integration von Software

In den siebziger und achtziger Jahren herrschten noch „Insellösungen“ auf den Landkarten betrieblicher Anwendungssysteme vor, also Softwareprodukte, die nur isoliert in einem Aufgabenbereich eingesetzt werden konnten. Seitdem fand nach und nach eine verstärkte Integration der Softwareprodukte statt, die zur sogenannten *integrierten Informationsverarbeitung* und entsprechenden Anwendungssoftwareprodukten wie dem R/3-System der SAP führte. Diese Art der Integration wird im Wesentlichen durch Datenbanksysteme mit zentralem Datenmanagement erreicht, auf denen alle Anwendungssysteme aufsetzten. Man spricht deswegen auch von *Datenintegration*. Durch eine darauf aufbauende *Funktionsintegration* werden permanent aggregierte und abgeleitete Daten automatisch fortgeschrieben. Durch den Einsatz des Internets nimmt die Integration neue Formen an, da durch die Softwareprodukte ebenfalls unternehmensübergreifende Geschäftsprozesse zwischen Geschäftspartnern (vielfältige Kunden-Lieferanten-Beziehungen) unterstützt werden müssen. Hier gerät die Datenintegration an ihre Grenzen, da mehrere

Unternehmen i. Allg. nicht mehr auf ein gemeinsames Datenbanksystem aufsetzen können. Die Funktionsintegration erfährt eine starke Herausforderung, da Datenbestände verteilter Unternehmen zumindest in Teilbereichen konsistent gehalten werden müssen. Diese und weitere technologische Fragen zur geschäftsmäßigen Zusammenarbeit mehrerer Organisationen auf Basis elektronischer Medien werden u.a. beim *Electronic Commerce* behandelt.

2.1.5 Aufgaben

Die Einsatzbereiche und damit die Funktionen betrieblicher Anwendungssysteme können anhand der Aufgaben eines Unternehmens gegliedert werden. So zählen Forschung und Entwicklung, Marketing, Beschaffung, Lagerhaltung, Produktion, Verkauf, Finanzbuchhaltung und Personalmanagement zu den typischen Aufgaben eines Industriebetriebs. In Abhängigkeit vom Industriezweig (Branche) und weitergehender betriebstypischer Unterscheidungsmerkmale kommen weitere Aufgaben und spezifische Verfahren hinzu.

Je nach Typ der unterstützten Aufgaben können betriebliche Anwendungssysteme in Administrations-, Dispositions-, Planungs- und Kontrollsysteme unterteilt werden (siehe Tabelle 2.1 und [Mertens et al. 1996]).

- *Administrationssysteme* dienen der Rationalisierung vorhandener Abläufe. So vereinfacht ein Personalverwaltungssystem die Verwaltung der Mitarbeiterdaten eines Unternehmens.
- *Dispositionssysteme* sollen die kurzfristige betriebliche Entscheidungsfindung vereinfachen bzw. übernehmen. Zu den Dispositionssystemen gehören u.a. Systeme zur Beschaffung, die für jedes gelagerte Teil (Roh-, Hilfs- und Betriebsstoffe, Halbfabrikate und Fertigfabrikate) je nach vorher gewählten Parametern (Mindestbestand, Wiederbeschaffungszeit und Verbrauchsgeschwindigkeit) geeignete Zeitpunkte und Mengen zur Nachbestellung dieser Teile ermitteln.
- *Planungssysteme* dienen der mittel- bis langfristigen Entscheidungsfindung und haben finanziell weitreichendere Auswirkungen als Dispositionssysteme. Sie ermöglichen i.d.R. das Erzeugen alternativer Pläne, die zur Findung möglichst guter Pläne untereinander oder zu Kontrollzwecken mit der Ist-Situation verglichen werden können. Dazu gehören auch Systeme zur Planung der Absatzmengen von Produkten und der Kapazitäten von Produktionsanlagen, Maschinen und Personal.
- *Kontrollsysteme* dienen Fach- und Führungskräften zum Erkennen von außergewöhnlichen und daher bemerkenswerten Datenkonstellationen. Sie greifen auf bestehende Daten zurück, um Planabweichungen aufdecken zu können. Dazu erhalten sie über Administrationssysteme Ist-Daten aus dem laufenden Betrieb, die sie mit den ursprünglich geplanten Daten von Dispositions- und Planungssystemen abgleichen.

Typ/Aufgabe	Produktion	Beschaffung/Lagerhaltung	Vertrieb	Personal
Administration	Betriebsdatenerfassung	Lagerverwaltung	Kundenverwaltung	Personalverwaltung
Disposition		Bestelldisposition	Tourenplanung	
Kontrolle	Fertigungsleitstand			
Planung	Absatz- und Kapazitäts-planung		Marketingplanung	Personaleinsatzplanung

Tabelle 2.1: Aufgaben und Typen von betrieblichen Anwendungssystemen (Bsp.)

Um komplexe Anwendungssysteme trotz ihrer zunehmenden Integration übersichtlich zu halten, werden sie anhand gängiger betrieblicher Aufgabengliederungen in fachliche Teilsysteme gegliedert. Die SAP gliedert ihr R/3-System beispielsweise dazu in sogenannte *Anwendungskomponenten*. Statt des Begriffes Komponente wurde in der Vergangenheit häufig die Bezeichnung *Modul* benutzt, durch die zunehmende Ausrichtung auf Komponententechnologien wird jedoch vermehrt der Begriff *Komponente* verwendet. Auf den Komponentenbegriff wird in Kapitel 3 noch genauer eingegangen. Er ist genereller als der Modulbegriff, der nur auf eine Programmiersprache ausgerichtet ist. Komponenten sollten hingegen programmiersprachenunabhängig einsetzbar sein.

Komponenten von Standardsoftware werden klassifiziert in

- funktionsbezogene Komponenten, die direkt den betriebswirtschaftlichen Aufgaben des Unternehmens dienen,
- funktionsübergreifende Komponenten, die allen Aufgaben zur Unterstützung dienen, und
- Systemkomponenten, die der technischen Realisierung funktionsbezogener und funktionsübergreifender Komponenten dienen.

Jede funktionsbezogene und funktionsübergreifende Komponente unterstützt eine Reihe betriebswirtschaftlicher Aufgaben, die aus technischer und logischer Sicht auch als *Transaktionen* bezeichnet werden.

2.2 Konfiguration von Standardsoftware

2.2.1 Begriffsdefinitionen

Betriebliche Standardsoftwareprodukte werden zu unternehmensindividuellen, d.h. kundenspezifischen Anwendungssystemen konfiguriert. Dazu werden vor der Nutzung des Anwendungssystems (zur *Definitionszeit*, genauer: Konfigurationszeit) Parameter auf ausgewählte Werte gesetzt. *Parameter* sind Datenfelder, die helfen, das Verhalten des

Systems zur Ausführungszeit (*Laufzeit*) entsprechend der Intention des Anwenders anzupassen. Je nach Art der zugrundeliegenden Softwarearchitektur können Konfigurationszeit und Laufzeit mehr oder weniger eng zusammenfallen. Im besten Fall kann sich der Anwender nach Änderung eines Parameters sofort auch auf dessen Auswirkungen verlassen und braucht ein Programm nicht erst neu zu initialisieren, zu starten oder gar durch Übersetzung der Programmtexte neu zu erzeugen. So wird beispielsweise im R/3-System der SAP ein interpretativer Ansatz bei der Berücksichtigung der Parameter verfolgt. D.h., Parameter, auch *Customizing-Daten* genannt, werden über eine Transaktion (*Customizing-Transaktion*) in die Datenbank eingestellt. Die betrieblichen Anwendungskomponenten, die von diesen Parametern abhängig sind, berücksichtigen (interpretieren) diese neuen Einstellungen beispielsweise bei Berechnungen oder bei der Auswahl von Verfahren (s.u.) sofort.

Hingegen werden in einem übersetzungsbasierten Ansatz die Parameter zur Steuerung der Übersetzung der Quellprogramme in ausführbaren Maschinencode genutzt. So können beispielsweise die Parameter in Softwareprodukten, die auf den Programmiersprachen C und C++ basieren, zur Steuerung eines Makro-Compilers genutzt werden, der dann den entsprechend der Benutzeranforderungen benötigten Programmquelltext zusammenstellt und diesen für die endgültige Übersetzung in Maschinencode vorbereitet.

Während der interpretative Ansatz typischerweise bei den heutigen Standardsoftware-produkten genutzt wird, wird der übersetzungsbasierte Ansatz vornehmlich beim Konfigurationsmanagement (siehe z.B. [Berlack 1992]) im Rahmen der eigentlichen Softwareentwicklung (beim Hersteller) verwendet.

Zur Unterscheidung der Begriffe Parametrierung und Konfiguration soll hier festgehalten werden, dass im Kontext dieser Arbeit Parametrierung als ein Teilaspekt der Konfiguration verstanden wird. *Parametrierung* ist die Einstellung von Werten an dafür vorgesehenen Stellen (Datenfeldern). Dabei können Parameterwerte durchaus komplexer Art sein, also z.B. ganze Softwarekomponenten darstellen. Die *Konfiguration* bezieht sich hingegen auf die Gestaltung eines komplexen Systems aus kleineren Komponenten und auf die Abhängigkeiten der Parameter zwischen den verschiedenen Komponenten des komplexen Systems.

2.2.2 Klassifizierung der Parametrierung

In diesem Abschnitt werden die Daten (Customizing-Daten), die der Parametrierung eines betrieblichen Anwendungssystems dienen, klassifiziert. Die Klassifizierung dient u.a. der genaueren Definition der prozessorientierten Konfiguration komponentenbasierter Anwendungssysteme, wie sie in dieser Arbeit erarbeitet wird.

Parameter sind Daten, die sich durch bestimmte Eigenschaften bzw. Eigenschaftsausprägungen von „normalen“ Daten wie Stamm- und Bewegungsdaten

unterscheiden. In [Dittrich, Mertens und Hau 1999] werden deshalb zunächst allgemeine Merkmale zur Klassifizierung von Daten betrieblicher Anwendungssysteme aufgestellt. Parameter stellen dann eine spezielle Klasse solcher Daten(typen) dar. Zu den allgemeinen Merkmalen und Merkmalsausprägungen wird folgende Liste angegeben.

- Die *Struktur* eines Datentyps sagt aus, ob es sich um ein Datenfeld*, einen Datensatz, eine Tabelle oder eine Datenbank handelt.
- Die *Funktion* eines Datums ist entweder planend*, dispositiv* oder administrativ wirksam.
- Die *Aufgabe* definiert die Art der Funktionserfüllung. Sie ist entweder steuernd*, berechnend*, auswählend*, identifizierend oder quantifizierend.
- Die *Stellung* sagt aus, ob es sich um ein einzugebendes*, ein referenzierendes oder ein auszugebendes Datum handelt.
- Die *Darstellung* eines Datums lässt sich grob in numerisch* (metrisch), alphabetisch* (nominal) und alphanumerisch* (nominal) gliedern.
- Der *Aufbau* des Datums ist entweder formatiert* oder unformatiert.

Die mit einem Stern („*“) gekennzeichneten Merkmalsausprägungen kennzeichnen die Parameterdaten. Für diese Arbeit ist aufgrund der Verhaltens- und Prozessorientierung das Merkmal „Aufgabe“ von Bedeutung. *Steuernd* bedeutet in diesem Kontext, dass ein Parameter der Auswahl einer Funktion aus einer Menge alternativ zur Verfügung stehender Funktionen dient. So kann ein Parameter beispielsweise festlegen, mit welchem Algorithmus ein Auftrag in einen bestehenden Auftragspool eines Produktionsplanungssystems eingeplant werden soll. In einem komponentenbasierten Ansatz würden die alternativen Algorithmen u.U. in Komponenten gekapselt werden und auf Basis einer standardisierten Schnittstelle gegeneinander austauschbar sein. Ein Parameter ist *berechnend*, wenn der Parameterwert als Konstante in die Berechnung einer Funktion einfließt. Ein einfaches Beispiel ist der Mehrwertsteuersatz bei der Berechnung von Verkaufspreisen. Ein Parameter ist *auswählend*, wenn er die Kriterien zur Auswahl der Input-Daten einer Funktion definiert. So können z.B. über einen solchen Wert Aufträge eines bestimmten Zeitintervalls (z.B. „von ‚jetzt‘ bis in 5 Tagen“) zur Planung ausgewählt werden. Berechnende oder steuernde Parameter würden in einem komponentenbasierten Ansatz u.U. als Parameter zur Konfigurationszeit zur Komponente angegeben werden. Andererseits könnte man sich auch vorstellen, dass solche Parameter wieder standardisiert über eine Schnittstelle von einer anderen Komponente übergeben werden.

Nach einer anderen Charakterisierung (siehe z.B. [Appelrath und Ritter 1999]) können Parameter strukturelle und verhaltensrelevante Aspekte des Anwendungssystems betreffen. Verhaltensrelevante Parameter bestimmen das Verhalten des Systems in Abhängigkeit der gewählten Parameter und haben somit eine steuernde Aufgabe. Ein einfaches Beispiel im R/3-

System der SAP ist die Preissteuerung bei der Bewertung und Kontierung von Materialien. Im Rahmen des Customizings kann festgelegt werden, ob Materialien einer Materialart

- per Voreinstellung als Vorschlag oder verbindlich (Parameter 1) und
- nach dem gleitenden Durchschnittspreis oder mit einem Standardpreis (Parameter 2)

bewertet werden sollen. Beim Anlegen eines Materialstamms kann in Abhängigkeit von Parameter 1 das für die Materialart voreingestellte Preissteuerungsverfahren überschrieben werden.

Zu den strukturellen Parametern zählen aufbauorganisatorische Parameter und Parameter, die sich auf Produkte (RHB-Stoffe, Erzeugnisse und Stücklisten) und Anlagen (Produktions-, Heizungsanlagen usw.) des Unternehmens beziehen. Sie dienen allgemein der Klassifizierung und Ordnung von Stamm- und Bewegungsdaten, die zur Laufzeit des Systems entstehen.

2.2.3 Konfigurationsalternativen

Anschaulich kann die steuernde Aufgabe (siehe 2.2.2) der Parametrierung – sie stellt, wie sich später noch zeigen wird, einen Hauptaspekt dieser Arbeit dar – als Konfiguration eines UND/ODER-Graphen betrachtet werden, dessen Verzweigungen und Knoten bestimmten Bedingungen genügen müssen. Als Beispiel dient in Abbildung 2.3 die MM-Komponente (Materials Management) des R/3-Systems der SAP. Dort ermöglicht ein Teilbereich der Logistik das Einrichten einer beliebigen Anzahl („1:n“) von Materialarten. Jeder Materialart kann eine oder keine Preissteuerung zugeordnet werden. Soll eine Preissteuerung zugeordnet werden, kann zwischen den in Abschnitt 2.2.2 genannten Alternativen (Durchschnitts- oder Standardpreis) gewählt werden. Ferner können bei der Konfiguration (Customizing) fast beliebig viele Warengruppen eingerichtet werden, über die z.B. später Auswertungen definiert werden.

Wie ein System konfiguriert werden muss, d.h. welche Strukturen und Verfahren in das System aufgenommen werden sollen, entscheiden i.d.R. die betroffenen Fachabteilungen. Die Einstellungen werden von der Fachabteilung unter Zuhilfenahme von Beratern und/oder der Informatikabteilung vorgenommen. Da ein Unternehmen sich aber veränderlichen Marktanforderungen und daraus resultierend auch veränderten Zielen anpassen muss, bleiben auch die Einstellungen nicht konstant und müssen an die neuen Begebenheiten angepasst werden. Aus diesem Grund sollten die Einstellungsmechanismen (Customizing-Transaktionen) so einfach zugänglich sein, dass sie durch die Fachabteilung selbständig – ggf. unter Absprache mit weiteren betroffenen Abteilungen – durchgeführt werden können.

Wichtig ist dabei auch, dass dem Anwender bzw. der Fachabteilung die Konsequenzen der Parameterwertauswahl verdeutlicht werden und nicht bloß auf das technische „Zum-Laufen-Bringen“ eines Anwendungssystems gezielt wird. In [Dittrich, Mertens und Hau 1999] wird

deswegen zurecht vor den betriebswirtschaftlichen Folgen eines falschen „*Management by Parameters*“ gewarnt. So können ungeschickt und gedankenlos gewählte Parameterwerte beispielsweise schnell zu einer wesentlich höheren Kapitalbindung führen. So werden einfache Beispiele genannt, die zu einer 20- bis 30%igen Steigerung der Kapitalbindung führen, worunter z.B. die Liquidität des Unternehmens leidet.

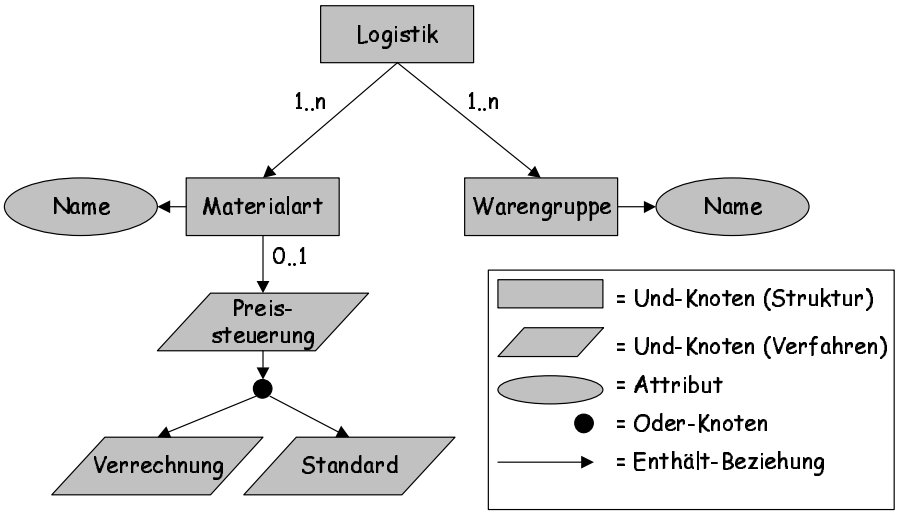


Abbildung 2.3: Konfigurationsbaum

Ein eher technisches Problem bei der Konfiguration ist die *Release-Fähigkeit*, also die Forderung, dass Anwendungssysteme, die einmal parametriert wurden, sich auch nach Updates/Upgrades durch den Hersteller noch genauso verhalten wie zuvor. Bei der Parametrierung marktgängiger Systeme wie dem R/3-System garantiert i.d.R. der Softwarehersteller die Release-Fähigkeit.

Bei monolithischen Systemen ist die Release-Fähigkeit damit für den Anwender ein weniger großes Problem als bei den in dieser Arbeit betrachteten komponentenbasierten Systemen, da die Schnittstellen zwischen den Komponenten eines komplexeren Systems zum Großteil in der Obhut eines einzigen Herstellers liegen.

2.3 Konfigurationshilfsmittel

Im Umfeld der Konfiguration bzw. des Customizings betrieblicher Anwendungssysteme wie dem R/3-System ist die Einführungsdauer der Systeme für ihre Hersteller sowie für die unterstützend zur Seite stehenden Berater ein entscheidender Faktor. So geriet beispielsweise die SAP Mitte der 90er Jahr zunehmend unter Druck, da viele R/3-Einführungen bei mittelständischen Firmen oft mehr als zwei Jahre dauerten und dabei große und vor allem teure Beraterteams an das Projekt banden. Aus diesem Grund entwickeln die Hersteller und Beratungsunternehmen Methoden und Werkzeuge, die die Einführung von betrieblichen

Anwendungssystemen beschleunigen sollen. In der Praxis haben sich bisher die Werkzeuge „Softwarereferenzmodell“, „Checkliste“ und „Vorkonfiguration“ bewährt.

2.3.1 Softwarereferenzmodelle

Vor dem Hintergrund der BPR-Idee (Business Process Reengineering, siehe [Hammer und Champy 1994]), bei der die geschäftsprozessorientierte, am Kunden ausgerichtete Gestaltung des Unternehmens besondere Beachtung findet, bekommen Softwarereferenzmodelle eine zunehmende Bedeutung. *Softwarereferenzmodelle* sind graphische Beschreibungen auf Basis von Diagrammsprachen, die die Strukturen (im Wesentlichen Datenstrukturen), Funktionen (*betriebliche Transaktionen*) und die dadurch unterstützten Geschäftsprozesse eines Anwendungssoftwareprodukts beschreiben. Sie dienen in erster Linie der Analyse der Potentiale eines Softwareproduktes. In späteren Phasen (Soll-Konzeption, Entwurf) lassen sich diese Beschreibungen nutzen, um die zukünftig mit einer neuen Software abzuwickelnden Geschäftsprozesse näher zu beschreiben. Ein angestrebtes – aber tatsächlich bisher unerreichtes – Ziel ist das „prozessorientierte Customizing“ auf Basis von Referenzmodellen. Es beruht auf der Idee, die Anwender selbst mittels Werkzeugunterstützung ihre unternehmensspezifischen Prozesse aus einem Katalog (Referenzmodell) auswählen und anpassen zu lassen. „Per Knopfdruck“ sollen sich dann die ermittelten Informationen (Prozesse und Strukturen) zur Konfiguration eines individualisierten Anwendungssystems nutzen lassen. Die Umsetzung dieser Idee ist bislang gescheitert. Zu den Gründen gibt es bisher keine fundierten Untersuchungen. Sie ist vermutlich gescheitert, weil

- die Komplexität der Konfiguration unterschätzt wurde,
- die zugrunde liegenden und heute verbreiteten Modelle den formalen Ansprüchen des Konfigurationsproblems nicht gerecht wurden (so wurde beispielsweise versucht, die Konfiguration auf Basis der semi-formalen Diagrammsprache EPK (Ereignisgesteuerte Prozesskette, siehe Abschnitt 4.2.2) zu etablieren) und
- die Gestaltung der Geschäftsprozesse bzw. die durch sie definierte Koordination der Arbeitsschritte innerhalb eines betrieblichen Anwendungssystems nur einen Teilaspekt der Gesamtkonfiguration ausmachen.

Bei der SAP (siehe [Appelrath und Ritter 1999]) wird deswegen mittlerweile ein modifizierter Weg eingeschlagen. Im Mittelpunkt stehen nicht mehr die eigentlichen Prozesse des Referenzmodells, sondern eine Prozesshierarchie, in der alle unterstützten Prozesse eingeordnet und top-down durch den Anwender durchsucht werden können. Es können dabei

nur Prozesse ausgewählt, in Hinblick auf die eigentliche Konfiguration jedoch nicht abgeändert oder neu hinzugefügt werden[1].

Prozesse und Softwarekomponenten stehen in einem unmittelbaren Verhältnis zueinander: Die einzelnen Schritte eines Geschäftsprozesses können nur unter Rückgriff auf verschiedene Softwarekomponenten ausgeführt werden. Das bedeutet andererseits, dass durch die Auswahl auch die benötigten Softwarekomponenten ausgewählt werden. Dieser Sachverhalt wird beim R/3-System genutzt, um zu ermitteln, welche Komponenten für den Einsatz des R/3-Systems noch konfiguriert werden müssen.

Im Rahmen dieser Arbeit wird vor dem Hintergrund der komponentenbasierten Anwendungssysteme ein neuer Vorschlag für die „prozessorientierte Konfiguration“ gemacht. Aus diesem Grund wird in Abschnitt 4.2.3 nochmals genauer auf Softwarereferenzmodelle eingegangen.

2.3.2 Checklisten

Eine *Checkliste* stellt ein Hilfsmittel dar, um den Konfigurationsraum eines standardisierten Softwareproduktes interaktiv, d.h. durch einen Frage-Antwort-Dialog zwischen Anwender und System systematisch top-down einzuschränken. So wird beispielsweise zu Beginn der Interaktion gefragt, welche Funktionsbereiche eines Unternehmens DV-Unterstützung durch das einzusetzende System erfahren sollen. Auf Basis der Antworten werden die benötigten Komponenten ausgewählt. Durch Abfragen weiterer betriebstypischer Merkmale, z.B. ob es sich beim einsetzenden Unternehmen um einen Einzelfertiger oder einen Massenfertiger handelt, können Rückschlüsse auf die benötigten Planungs- und Kalkulationsverfahren gemacht werden. Der Anwender wird systematisch durch alle relevanten Fragen gelenkt. Durch die Auswertung seiner Antworten kann nach und nach der Konfigurationsraum eingeschränkt werden. Logische Abhängigkeiten zwischen verschiedenen Parametern – aus Sicht eines formalen Konfigurationsmodells könnte man von Constraints sprechen – können durch ein solches System automatisch berücksichtigt werden, indem entsprechende Propagierungsmechanismen implementiert werden. Aus der Praxis ist das System R/3 LIVE KIT Structure bekannt (siehe [IBIS Prof. Thome GmbH 1999]), das den Konfigurationsaufwand des R/3-Systems einschränkt.

Ein Problem dieser Werkzeuge ist, dass das Konfigurationswissen zweifach vorhanden ist, einmal im Anwendungssoftwareprodukt, das andere Mal im Konfigurationssystem. Die Informationen müssen logisch konsistent sein. Zudem stellt sich die Frage, welche Worte in der Checkliste verwendet werden soll. Wird die Terminologie des Anwendungssystems

---

[1] Natürlich können neue Prozesse definiert werden. Sie haben jedoch keinerlei automatisierte

gewählt, die für den Anwender noch unbekannt bzw. anders belegt ist, dann werden oft Erwartungen bzgl. der Anforderungsabdeckung geweckt, die durch das System u.U. tatsächlich gar nicht erfüllt werden. Wird eine eigenständige Terminologie gewählt, muss sich der Anwender neben der Begriffswelt des Anwendungssystems und der des Unternehmens noch in eine dritte hineindenken. In [Dittrich, Mertens und Hau 1999] wird zudem kritisiert, dass die Auswirkungen der getroffenen Parametereinstellungen – selbst bei einem grundlegenden Verständnis des Systems – kaum bewertet werden können. Hier könnten entsprechende Erklärungskomponenten Abhilfe schaffen, die jedoch in diesem Aufgabenbereich bislang nur als Forschungsprototypen existieren.

2.3.3 Vorkonfigurierte Systeme

Das wiederkehrende Problem der Konfiguration kann man auch dadurch versuchen zu umgehen, dass man vorkonfigurierte (*pre-customized*) Systeme – auch Referenzsysteme genannt – anbietet. So ist es denkbar, branchen- und/oder betriebstypische Konfigurationen der Softwareprodukte oder Teile davon vorzuhalten. Wie in [Dittrich, Mertens und Hau 1999] angemerkt wird, können diese Systeme der Simulation verschiedener Geschäftsprozesse dienen und das System für den Anwender durch den Beispielcharakter verständlicher machen. Jedoch zeigt die Praxis, dass das anwendende Unternehmen nicht umhin kommt, alle Parametereinstellungen bzgl. der unternehmenseigenen Anforderungen auf Validität zu prüfen. Weiterhin ist anzumerken, dass die Wechselwirkungen verschiedener Teilkonfigurationen, also von Konfigurationen, die nur einen Teil (Komponente, Modul) des Systems betreffen, aus formaler Sicht besonderer Berücksichtigung bedürfen. So kann man sich leicht vorstellen, dass Konfigurationsinformationen zu einer Komponente im Widerspruch zu Konfigurationsinformationen einer anderen Komponente stehen und dass die Informationen bei sukzessive erfolgendem Einspielen gegenseitig überschrieben werden. Entsprechende Automatismen, die Konflikte aufdecken und im Zweifelsfall den Anwender entscheiden lassen, welche Einstellungen zu bevorzugen sind, werden zunehmend bereitgestellt.

2.3.4 Weitere Ansätze

In [Dittrich, Mertens und Hau 1999] werden noch ein ereignisgesteuertes, wissensbasiertes System (E-WBS) und ein Simulationssystem vorgestellt. Das E-WBS dient der Kontrolle der Parametereinstellungen zur Laufzeit. Mittels des Simulationssystems können Auswirkungen der getroffenen Einstellungen bewertet werden, indem das resultierende Verhalten des Anwendungssystems unter vorher getroffenen Annahmen (z.B. Anzahl und Art der eintreffenden Aufträge innerhalb einer bestimmten Periode) exemplarisch durchgespielt wird.

Auswirkungen auf den Konfigurationsvorgang.

Beide Ansätze sind also keine Konfigurationshilfen im eigentlichen Sinne. Sie veranschaulichen vielmehr die Konsequenzen, die sich aus den Parametereinstellungen ergeben.

2.3.5 Konsequenz

In der heutigen Praxis spielen unterstützende Konfigurationswerkzeuge immer noch eine untergeordnete Rolle. Ein Problem ist, dass die Softwareprodukte und die zugehörigen Beschreibungen (z.B. Softwarereferenzmodelle) nicht ausreichend harmonieren. Softwarereferenzmodelle, wie sie durch das R/3-Referenzmodell der SAP bekannt geworden sind, sind nachträglich erstellte Beschreibungen, die nicht mit der Struktur der Software übereinstimmen.

Es ist offensichtlich, dass die Konfiguration nicht ohne eine entsprechende Unterstützung effizient durchgeführt werden kann. Im Rahmen dieser Arbeit wird ein Ansatz vorgestellt, der Teilaspekte der sich als praktikabel herausgestellten Hilfsmittel aufgreift. So werden vor allem die Grundideen der prozessorientierten Sichtweise auf Softwaresysteme durch Softwarereferenzmodelle aufgegriffen.

3 Komponentenbasierte Anwendungssysteme

Im vorherigen Kapitel wurde dargelegt, dass heute bei der Gestaltung betrieblicher Anwendungssysteme vornehmlich monolithische Standardsoftwareprodukte zum Einsatz kommen. Eine Annahme (siehe Abschnitt 1.4), die dieser Arbeit zugrunde liegt, geht jedoch davon aus, dass früher oder später ein Markt für flexiblere, komponentenbasierte Softwareprodukte entsteht. Aus diesem Grund werden die Eigenschaften der sich heute abzeichnenden komponentenbasierten Ansätze für betriebliche Anwendungssysteme näher beschrieben. Zuerst werden dazu in Abschnitt 3.1 ein paar Grundlagen aufbereitet, die sich im Wesentlichen aus der Historie begründen, die diesen Systeme voranging. In Abschnitt 3.2 wird eine Definition von Komponenten und eine Abgrenzung gegenüber anderen bekannten Ansätzen wie objektorientierten und Modulkonzepten vorgenommen. Im nächsten Abschnitt (3.3) wird auf die Schnittstellenproblematik eingegangen. Im letzten Abschnitt (3.4) wird die Problematik des Softwarekonfigurationsmanagement angegangen, die sich bei komponentenbasierten Systemen aufgrund ihrer Struktur nicht nur bei der Komponentenentwicklung, sondern in besonderem Maße auch bei der Komposition und Konfiguration von Anwendungssoftwareprodukten stellt.

3.1 Grundlagen

Komponentenbasierte Anwendungssysteme stellen einen Mittelweg zwischen dem Einsatz monolithischer Standardsoftwareprodukte und individuell entwickelter Systeme dar. Eine für Softwarekomponenten oft gewählte Metapher ist die des *Bausteins*, wie sie auch in anderen ingenieurwissenschaftlichen Disziplinen benutzt wird. Sie suggeriert, dass sich Bausteine relativ einfach zum Aufbau größerer Systeme (Häuser, Maschinen, Anlagen, Autos usw.) nutzen lassen. Softwarekomponenten sind dann in diesem übertragenen Sinne Bausteine, die ein „Programmieren im Großen“ ermöglichen. Allerdings gibt es auch entscheidende Unterschiede zwischen diesem herkömmlichen Verständnis von Bausteinen und dem der Softwaretechnik. Beispielsweise sind Softwarebausteine immateriell und können mehrfach instantiiert werden, und meist sind ihre Schnittstellen wesentlich komplexer. Was „gute“ Komponenten bzw. aus ihnen aufgebaute „gute“ Systeme sind, ist bislang weder in der Praxis noch von theoretischer Seite hinreichend thematisiert, geschweige denn geklärt. Ein Problem dabei ist, dass viele Begriffe verschiedener Teildisziplinen der Softwaretechnik fälschlicherweise gleichgesetzt, verwechselt oder mehrdeutig definiert werden. Oft werden beispielsweise komponenten- mit objektorientierten Konzepten gleichgesetzt: Unter einer Komponente wird demnach nichts anderes als ein Objekt oder ein Klasse verstanden. Deshalb wird in [Szyperski 1998] explizit angemerkt, dass objektorientierte Softwareentwicklung nicht notwendigerweise zu Komponenten führt und komponentenbasierte Systeme nicht notwendigerweise objektorientierte Entwicklungssprachen voraussetzen. Im Folgenden sollen verschiedene Kategorien wie Objekte (bzw. Klassen), Komponenten, Verteilung usw. deshalb

klar voneinander unterscheiden werden. Die Grundlagen bilden das Fundament für das Komponentenmodell, das in dieser Arbeit konstruiert wird.

3.1.1 Objektorientierung

Auch wenn zu Beginn des Abschnitts 3.1 herausgestellt wurde, dass Komponenten nicht notwendigerweise objektorientierte Softwaretechnologie voraussetzen, stellt die Objektorientierung doch einen der Meilensteine der Entwicklung in diesem Bereich dar, denn objektorientierte Softwaretechnologie vereinfacht die Entwicklung komponentenbasierter Softwaresysteme entscheidend. Aus diesem Grund soll kurz auf ihre wichtigsten Aspekte eingegangen werden.

In der objektorientierten Softwaretechnik geht man davon aus, dass sich alle für Anwendungssysteme relevanten Sachverhalte als Objekte darstellen lassen. Aus Objekten können Objektsysteme (Familien von Objekten) gebildet werden, in denen die Objekte Beziehungen zueinander haben und miteinander kommunizieren. Objekte sind typisiert, d.h. alle Objekte eines Typs (Objekttyp) besitzen gemeinsame Eigenschaften (Attribute wie Farbe, Höhe, Breite, Preis) und gleiche Fähigkeiten, in der objektorientierten Welt *Methoden* genannt. So können beispielsweise Lieferanten „Materialien liefern“ und „Rechnungen schicken“. Weitere, darauf aufbauende Kernprinzipien objektorientierter Systeme sind

- die *Objektidentität*, die zusichert, dass jedes Objekt über einen Bezeichner (ObjectId) eindeutig zu identifizieren ist,
- die *Vererbung* (Schnittstellenvererbung, siehe Abschnitt 3.1.2), durch die die Fähigkeiten und Eigenschaften einer Definition eines Objekttyps (Supertyp) an die Definition eines anderen Objekttyps (Subtyp) weitergegeben (vererbt) werden können,
- die *Polymorphie*, die es ermöglicht, dass die vergleichbare Fähigkeiten von verschiedenen Subtypen eines gemeinsamen Supertyps unter ein und demselben (Fähigkeiten-)Bezeichner (dem Namen der Methode) abgerufen werden können,
- die *Kapselung* von Eigenschaften und zugehörigen Fähigkeiten innerhalb eines Objekttyps und
- das *Information Hiding* – die Möglichkeit, Implementierungsdetails vor dem Nutzer (Client) eines Objekts (Server) zu verbergen.

3.1.2 Subtyping

Fast alle Programmiersprachen (Sprachen der vierten Generation (4GL), Cobol, C/C++, Java), die heute zur Realisierung von betrieblichen Anwendungssystemen genutzt werden, sind typisiert. Eine Programmiersprache ist *typisiert*, wenn in Programmen dieser Sprache die Datentypen der verwendeten Variablen, Konstanten, Funktionen usw. zur *Deklarationszeit*

festgelegt sein müssen. Ein *Datentyp* ist die Zusammenfassung von semantisch ähnlichen Wertebereichen und Prozeduren auf diesen Wertebereichen zu einer Einheit. Vorteile der Typisierung sind, dass Laufzeitfehler wie Speicherzugriffsverletzungen von vornherein ausgeschlossen werden können und dass eine „Normierung“ für eine arbeitsteilige Softwareentwicklung erreicht werden kann. *Sub(daten)typen* sind Datentypen, deren Wertebereiche eine Teilmenge eines übergeordneten Datentyps (Supertyp) darstellen, die vergleichbare Prozeduren wie die (potentiellen) Supertypen bereitstellen, und zudem überall dort einsetzbar sind, wo auch der originäre Datentyp verwendet werden kann. Man kann mindestens drei Arten von Subtypbeziehungen unterscheiden:

- Bei der *Schnittstellenvererbung* (*Interface Inheritance*), die in den meisten objektorientierten Programmiersprachen angewandt wird, wird ein Subtyp durch explizite Nennung einer Reihe von Supertypen, von denen er die Signaturen[1] (evtl. sogar die Implementierung) der Operationen (Methoden) erbt, definiert. Zusätzlich können in dem Subtyp noch neue Operationen(-signaturen) definiert werden.
- Bei der strukturellen Subtypbeziehung (*structural subtyping*) werden die Supertypen nicht explizit angegeben. Die Tatsache, ob ein Datentyp Subtyp eines anderen Datentyps (Supertyp) ist, wird ausschließlich darüber ermittelt, ob die Menge seiner Methodensignaturen Teilmenge der Signaturen des Supertyps ist.
- Die verhaltensbezogene Subtypbeziehung (*behavioural subtyping*) schließt das Verhalten eines Datentyps – d.h. vereinfacht gesagt: die erlaubten Sequenzen von Methodenaufrufen (Traces, siehe Abschnitte 3.3.2 und 7.4) – mit in die Subtyprelation ein. Ein Datentyp kann nur dann Subtyp eines anderen Datentyps (Supertyp) sein, wenn er alle Kommunikationssequenzen erlaubt, die auch der Supertyp erlaubt.

Offensichtlich ist das Behavioural Subtyping nur dann gegeben, wenn auch die Methodensignaturen des Supertyps in dem Subtyp vorkommen, da sonst gleiche Kommunikationssequenzen nicht möglich sind. Daraus folgt, dass zusätzlich eine der beiden ersten Subtypbeziehungen gelten muss.

Bei der Wiederverwendung von Programmelementen spielen Subtypbeziehungen offensichtlich eine tragende Rolle, denn durch die Normierung von Schnittstellen, wie sie durch Subtyping erreicht wird, kann eine Arbeitsteilung erreicht werden, die ja gerade vornehmliches Ziel der Wiederverwendung ist. Ein Element kann nur dann wiederverwendet

---

[1] Die Signatur bezeichnet eine Operation (Methode oder Prozedur) innerhalb einer Klasse, einem Modul oder einer Komponente eindeutig. Sie setzt sich aus ihrem Namen sowie den Typen und den Positionen der einzelnen Argumente der Methode/Prozedur zusammen.

werden, wenn es die erwarteten Schnittstellen bereitstellt, also Subtyp eines vorgegebenen Datentyps ist. Während die Konformität der Signaturen relativ einfach automatisch geprüft werden kann, stellt der Nachweis der Konformität des Verhaltens ein größeres Problem dar, auf das in den Abschnitten 4.1.2 nochmals eingegangen wird.

3.1.3 Verteilung

Gerade die objektorientierte Integration von Eigenschaften und Fähigkeiten, oft als *Struktur* und *Verhalten* bezeichnet, legt die Betrachtungsweise nahe, Objekte im Sinne des Client/Server-Modells als kleinste mögliche Einheiten (Prozesse) für die Verteilung auf Rechnersysteme zu betrachten. Jedes Objekt nutzt in der Rolle des Clients Fähigkeiten anderer Objekte und kann dritten Objekten seine Fähigkeiten als Server zur Verfügung stellen.

Während in den klassischen *Drei-Ebenen-Architekturen*[1] die gesamte Anwendungsebene aus einem oder zumindest aus einer fest vorgegebenen Anzahl von Server-Prozessen besteht, lässt sich durch objektorientierte Technologien diese Ebene in eine Reihe von vernetzten Client- und Serverobjekten aufteilen, die zudem physisch fast beliebig auf Rechnersysteme verteilt werden können. Objekte interagieren miteinander über Rechnergrenzen hinweg und können andere Objekte dazu veranlassen, ihre Fähigkeiten (Methoden) auszuüben. Der Entwickler eines objektorientierten Systems muss sich im Rahmen der Entwicklung bei modernen Verteilungsarchitekturen nicht darum kümmern, auf welchem Rechner sein Objekt „liegt“. In den klassischen Drei-Ebenen-Architekturen dagegen werden zwischen den Anwendungen Nachrichten verschickt, die auf der Empfängerseite zentral empfangen werden und den Empfängerobjekten (Funktionen) dann erst zugeordnet werden müssen. Der Entwickler muss dazu meist angeben, auf welchem Rechner sich das Empfängerobjekt befindet.

Die transparente Verteilung von Objekten über Rechnergrenzen hinweg wird durch moderne Architekturen wie der *Common Object Request Broker Architecture* (*CORBA,* siehe [Yang und Duddy 1996]) von der *Object Management Group (OMG)* und dem *Distributed Common Object Model* (*DCOM*, siehe [Tallman und Kain 1998]) von Microsoft unterstützt. Diese Architekturen standardisieren nur die Art und Weise der verteilten Kommunikation, jedoch

---

[1] Die Drei-Ebenen-Architektur ist eine etablierte Client/Server-Architektur, die in den meisten Standardsoftwareprodukten wie Oracle Applications oder SAP R/3 (siehe [Appelrath und Ritter 1999]) zugrunde gelegt wird. Sie besteht aus den Ebenen Präsentationsebene, Anwendungsebene und Datenbankebene. Zwischen Präsentations- und Anwendungsebene sowie zwischen Anwendungs- und Datenbankebene bestehen Client/Server-Beziehungen. Alle Ebenen können verteilt auf verschiedenen Rechnern betrieben werden.

nicht den Inhalt der Kommunikation, also welche Daten/Objekte in welchen Formaten ausgetauscht werden.

3.1.4 Verhältnis von Objekten und Prozessen

In objektorientierten Programmiersprachen wird meist zwischen den beiden Konzepten Objekt und Prozess unterschieden. Unter *Prozessen* werden Betriebssystemprozesse verstanden, die entweder eigenständige Programme und/oder „leichtgewichtige“ Prozesse (sogenannte *Threads*) innerhalb eines Programms sind und untereinander kommunizieren können. Beispielsweise gibt es in der Programmiersprache Java (siehe [Flanagan 1997]) explizit eine Klasse `Thread`, die der Verwaltung von „leichtgewichtigen“ Prozessen dient. Während in diesen Sprachen die Konzepte „Objekt“ und „Prozess“ orthogonal zueinander stehen, werden in anderen Ansätzen (z.B. ACTORS, siehe [Agha und Hewitt 1987], [Honda und Tokoro 1991] und [Walker 1995]) Objekte grundsätzlich als eigenständige Prozesse, sogenannte *Aktoren*, verstanden. Die Grundlage für diese oft formal fundierten Ansätze bilden i.d.R. Transitionssysteme oder Prozesskalküle. Um beiden Ansätzen gerecht werden zu können, wird in dieser Arbeit (siehe Abschnitt 3.4) ein Ansatz verfolgt, der eine flexible Zuordnung von Objekten zu Prozessen erlaubt.

3.1.5 Klassifikation von Interaktionen

Interaktion kann als aufeinander bezogenes Handeln zweier oder mehrerer Objekte – in dieser Rolle werden sie dann auch als Akteure bezeichnet – verstanden werden. Im übertragenen Sinne wird auch bei objektorientierten oder komponentenbasierten Systemen von Interaktionen gesprochen. Korrekter wäre es, von *Kommunikationen* zu sprechen, da es sich ausschließlich um ein sprachliches (bzw. symbolisches) Handeln dreht. In dieser Arbeit werden die Begriffe, wie heute in Forschung und Praxis üblich, synonym gebraucht.

Kommunikationen (Interaktionen) lassen sich nach verschiedenen Kategorien klassifizieren:

- Zeit: synchron versus asynchron,
- Raum: lokal versus verteilt,
- Anzahl der Akteure: bilateral versus multilateral,
- Art und Weise (z.B. Frage-Antwort-Dialog, Befehl, Aussage).

3.2 Komponenten

„Komponenten dienen der Komposition“. Dieses sinngemäß übersetzte Zitat aus [Szyperski 1998] klingt selbstverständlich, stellt aber schon ein wesentliches Merkmal von Komponenten heraus: Die Komposition ermöglicht es, „vorfabrizierte Dinge“, eben Komponenten, in verschiedenen Systemen wiederzuverwenden, indem man sie nach bestimmten Regeln mit

anderen „Dingen“ (Komponenten) zusammensetzt. Diese Art der Wiederverwendung trifft jedoch auf viele Produkte der Softwareentwicklung zu, beispielsweise Makros und Templates, wie sie in Programmiersprachen wie C und C++ zu finden sind. Deshalb soll eine erste, konkretere Definition für Komponenten gegeben werden.

3.2.1 Definition von Komponenten

Unter *Komponenten* werden – angelehnt an [Szyperski 1998] – Softwareprodukte verstanden, die

- in binärer und damit – mit Ausnahme von Parametrierungsaspekten – unveränderlicher Form vorliegen,
- vertraglich zugesicherte Schnittstellen einhalten,
- von unterschiedlichen Produzenten unabhängig voneinander entwickelt werden können,
- durch Dritte unabhängig voneinander ein- bzw. zusammengesetzt werden können und
- zusammen mit anderen Komponenten ein vollständiges Anwendungssystem ergeben können.

Der erste Punkt schränkt Komponenten auf die Softwareprodukte ein, die für den Nutzer eine Black Box darstellen. Die anderen Punkte betonen die arbeitsteilige Vorgehensweise, die der komponentenbasierten Softwareentwicklung zugrunde liegt. So impliziert der zweite Punkt beispielsweise, dass eine Komponente sehr gründlich von ihrer Umgebung isoliert sein sollte. Ausschließlich über die definierten Schnittstellen weist sie Abhängigkeiten zu anderen Komponenten(-Frameworks) auf. Ein Entwickler, der diese Komponente einsetzen möchte, bekommt i.d.R. keine Informationen über die Art und Weise der Implementierung dieser Komponente. Er erhält ausschließlich Informationen, die zum Einsatz der Komponente bzw. zur Komposition notwendig sind: Dazu gehören Informationen über die Schnittstellen, das Verhalten und u.U. über nicht funktionale Aspekte wie z.B. die Performanz einer Komponente.

Beispiele komponentenbasierter Systeme sind folgende:

- Betriebssysteme bieten einen Rahmen für die Ausführung und Kommunikation – beispielsweise über Pipes oder TCP/IP-Sockets – verschiedener Komponenten (hier: Anwendungsprogramme).
- Microsofts Visual Basic in Verbindung mit der OLE-Technologie stellt eine weitere Umgebung für das Einbetten und Verknüpfen (Scripting) von Komponenten – in diesem Fall OLE- bzw. ActiveX-Controls – dar.
- Der Netscape Navigator ist ein Anwendungsprogramm, das anderen Komponenten, sogenannten Plug-Ins, als Ausführungsumgebung dient.

Komponentenbasierte Systeme können auch hierarchisch über mehrere Ebenen aufgebaut werden. So läuft der Netscape Navigator innerhalb eines Betriebsystems und beherbergt wiederum eine Reihe von Plug-Ins. Ein Ziel bei der Entwicklung moderner mehrstufiger Komponentenarchitekturen ist, die Schnittstellentechnologie über alle Stufen zu standardisieren, um letztlich „beliebige“ Hierarchien zuzulassen. So könnte es durchaus sinnvoll sein, ein Anwendungsprogramm als Plug-In in Netscape oder ein Plug-In als Anwendungsprogramm unter einem Betriebssystem laufen zu lassen.

3.2.2 Klassen- und Komponenten-Frameworks

Im Bereich des Software Engineerings gab es bisher unterschiedliche Ansätze, die Wiederverwendung von Softwareprodukten zu verbessern. Die Konzepte zur Wiederverwendung haben sich vornehmlich auf den Prozess der Softwareentwicklung im engeren Sinne, also auf den Entwurf und die Programmierung konzentriert. Erste Schritte (siehe z.B. [Biggerstaff und Richter 1987]) bestanden in der Entwicklung von *Funktions*- und *Klassenbibliotheken*. Im zweiten Schritt wurde das strikte Prinzip des einseitigen Aufrufs der Funktionen/Methoden einer Bibliothek durch eine Applikation aufgehoben: Sogenannte *Callback-Verfahren* wurden entwickelt, die erlauben, dass eine Applikation sich in den Informationsfluss einer Klassenbibliothek „einklinkt“ und bei wichtigen Ereignissen automatisch benachrichtigt wird. Dieses Verfahren wird vor allem bei GUI-Klassenbibliotheken und sogenannten Anwendungs-Frameworks ausgenutzt, um Aktionen, die von außen, also von anderen Systemen wie z.B. dem Betriebssystem angestoßen werden, an die eigentliche Anwendung zu melden. Neben den eher imperativ ausgerichteten Callback-Verfahren haben sich mit zunehmender Objektorientierung sogenannte Klassen-Frameworks etabliert. *Klassen-Frameworks* – auch objektorientierte Frameworks genannt (siehe [Mattsson 1996]) – sind spezielle Klassenbibliotheken, die vordefinierte abstrakte Klassen – sogenannte *Hot Spots* – definieren, die ein Anwendungsentwickler auf Basis von Vererbungmechanismen zur Implementierung von anwendungsspezifischen Erweiterungen nutzen kann. Klassen-Frameworks basieren auf sogenannten *Entwurfsmustern* (siehe [Gamma et al. 1995]), die typische Objekt- und Klassenstrukturen auf Grundlage abstrakter Klassen beschreiben.

Ein *Framework* ist also ein „noch nicht fertiges“ Softwareprodukt, das als Grundlage für eine Klasse (*Produktfamilie*, engl. *product line*) konkreter Anwendungssoftwareprodukte dient. Es besteht aus einer vordefinierten Softwarearchitektur, die in gewissen Grenzen durch zusätzliche Softwareelemente erweitert und dann wie oben unter dem Begriff „Verteilung“ beschrieben auf verteilten Rechnersystemen ablaufen kann. Entscheidend an diesen Systemen ist, dass sie neben einer Reihe festgelegter Aufrufschnittstellen ein definiertes Verhalten voraussetzen, das durch die zu ergänzenden Softwareelemente eingehalten werden muss. Diese Schnittstellen und das Verhalten eines Frameworks werden in einem „Vertrag“ (*Kontrakt*) definiert und sind für die zusätzlichen Softwareelemente verbindlich, da ansonsten das Gesamtsystemverhalten nicht vorhersehbar ist.

Ein Beispiel für ein Klassen-Framework[1] ist das *San Francisco* von IBM (siehe [Abinava et al. 1998]), das eine allgemeine Grundlage für betriebliche Anwendungssysteme bildet, die durch Klassen weiterer Softwareunternehmen (Independent Software Vendors, ISV) zu branchentypischen Systemen weiterentwickelt werden. IBM wird dieses Klassen-Framework zu einem Komponenten-Framework (siehe Abschnitt 1.2) ausbauen, indem es das Komponentenmodell *Enterprise JavaBeans* von SUN (siehe [Matena und Hapner 1999]) stärker berücksichtigt.

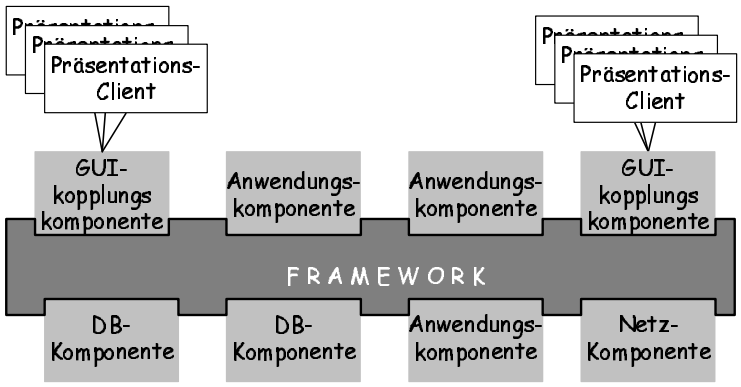


Abbildung 3.1: Framework

Nachteil der bisher vorgestellten Verfahren (Callback und Klassen-Framework) ist, dass die Basissysteme, also die Bibliotheken und Frameworks, dem Anwendungsentwickler nur als nützliche, aber nur in einem abgeschlossenen Kontext verfügbare Basis dienen, aber nicht als Plattform für unabhängig entwickelte und frei am Softwaremarkt angebotene Komponenten dienen können. Die Wiederverwendung bezieht sich in diesen Fällen ausschließlich auf die Bibliothek bzw. den Framework, aber nicht auf die an sie gekoppelten Systemerweiterungen. Zunehmend (siehe [Weck 1997], [Szyperski 1998]) wird aber eine grundsätzliche Erweiterung der Wiederverwendung gefordert, indem man größere Anwendungssysteme aus frei auf dem Softwaremarkt angebotenen Komponenten zusammensetzten können will. Solche komponentenbasierten Anwendungssysteme – in [Szyperski 1998] wird von *Component Software* gesprochen – entstehen nicht isoliert oder nur auf Grundlage eines herkömmlichen Betriebssystems. Sie brauchen vielmehr eine gewisse Infrastruktur, in der sie eingesetzt werden können. Diese Infrastruktur wird dann als *Komponenten-Framework* bezeichnet. Solche Frameworks sind durch Komponenten erweiterbare Softwaresysteme, die nicht primär die Vererbung (auf Basis abstrakter Klassen) unterstützen, sondern deutlich

[1]Die Firma IBM würde den Klassen-Framework SanFrancisco sicherlich schon heute als Komponenten-Framework bezeichnen. Tatsache ist, dass bislang kein richtiges Komponentenmodell unterstützt wird. Statt dessen müssen im Rahmen der Anwendungsentwicklung einzelne Java-Klassen – und nicht etwa Java-Packages – entwickelt und an das Framework angebunden werden

darüber hinausgehen, indem sie Basisdienste wie beispielsweise Retrieval-, Kommunikations-, Transaktions-, Sicherheits- oder Verteilungsdienste anbieten, die von den Erweiterungskomponenten genutzt werden können. Komponenten-Frameworks definieren also die grundlegende Architektur für die einzusetzenden Komponenten. Komponenten, die innerhalb eines Komponenten-Frameworks verwendet werden, können selbst wieder Frameworks sein, wodurch ein rekursives Entwurfsprinzip (*rekursive Erweiterung*) unterstützt wird.

Komponenten-Frameworks bieten i.d.R. verschiedene Erweiterungsmöglichkeiten, die unabhängig voneinander sind. So kann man sich vorstellen, dass ein Framework auf der einen Seite Komponenten für graphische Visualisierungen und auf der anderen Seite Komponenten zur persistenten Speicherung von Daten aufnehmen kann. Diese beiden erweiterbaren Eigenschaften stellen offensichtlich unterschiedliche Dimensionen der Erweiterbarkeit („dimensions of independent extensibility“, [Weck 1997]) dar. Sie sind in diesem Sinne *orthogonale Erweiterungen*. Im Gegensatz dazu stehen *parallele Erweiterungen*, die einen Framework gleichzeitig in Richtung einer einzigen Dimension erweitern. So könnte ein Framework beispielsweise mehrere Speicherungskomponenten aufnehmen.

3.2.3 Standards

Ein grundlegendes Problem bei der Komposition von Komponenten ist, dass die Komponenten aus verschiedenen Quellen (Herstellern) kommen und durch Dritte zusammengesetzt werden müssen. Deshalb bedarf es Standards, die es erlauben, Komponenten miteinander zu „verdrahten“ (*Wiring Standards*). Zu diesen Standards gehören heute Pipes unter Unix und Sockets im Bereich der heute vorherrschenden TCP/IP-Kommunikation. Darauf aufbauend haben sich mächtigere Standards wie Microsofts DCOM oder CORBA von der OMG etabliert. Diese Standards sagen jedoch nichts über die Inhalte aus, die zwischen Komponenten ausgetauscht werden sollen. Zunehmend wichtiger werden deswegen anwendungsbezogene Schnittstellenstandards, denn erst, wenn auf Seiten der Anwendungsdomäne Übereinkunft herrscht, welche Informationen in welchem Format ausgetauscht werden, können die Komponenten tatsächlich „semantisch verträglich“ kommunizieren und kooperieren. Aus welchen aktuellen Initiativen heraus sich ein solcher Standard durchsetzt, bleibt abzuwarten und ist aufgrund „politischer“ und taktischer Einflüsse schwer abzuschätzen. So gibt es auf der einen Seite herstellerübergreifende Standardisierungskomitees wie die *Open Application Group (OAG)* oder die Object Management Group, die inhaltliche Standards in Form sogenannter Business Documents oder Business Objects entwickeln. Durch die Vielfalt der Hersteller und damit verbundener Vorschläge blähen sich diese Standards unweigerlich auf, so dass die praktische Einsetzbarkeit in Frage gestellt werden kann. Andererseits könnten sich die Ansätze der marktbeherrschenden Unternehmen wie z.B. SAP im Bereich der betrieblichen Anwendungssysteme als De-facto-Standards etablieren.

3.2.4 Komponenten versus Objekte

Wie schon in Abschnitt 3.1 angedeutet, sind Komponenten und Objekte orthogonal zueinander stehende Konzepte. Die Unterschiede sollen hier noch weiter verdeutlicht werden.

Neben den in Abschnitt 3.2.1 betonten Eigenschaften wird in [Szyperski 1998] insbesondere betont, dass Komponenten Softwareelemente sind, die keinen persistenten, also über ihre Laufzeit hinweg dauerhaften Zustand besitzen. Diese Forderung impliziert, dass eine Komponente nicht von einer Kopie von sich selbst zu unterscheiden sein darf. Diese Forderung hat u.a. den Vorteil, dass in einer Laufzeitumgebung immer nur eine Instanz einer Komponente benötigt wird. Die Forderung nach Transienz bedeutet nicht, dass es nicht unterschiedliche Parametrierungen dieser Komponente geben darf. Jedoch darf davon nicht die *Funktionalität* betroffen sein, sondern nur die *Funktionsweise*. Die Parameterwerte können bzw. müssen sogar persistent sein, um sie zur Laufzeit verfügbar machen zu können. Sie gehören jedoch nicht zur Komponente, sondern sind vielmehr Bestandteil der Konfigurationsinformationen eines Anwendungssystems. Parameterwerte werden oft in sogenannten Ressourcendateien abgespeichert. Durch die Trennung der Konzepte „Komponente“ und „Konfiguration“ wird auch das Problem der Release-Fähigkeit von Komponenten entkräftet, da Konfigurationen i.d.R. langlebiger als Komponentenversionen sind.

In objektorientierten Komponenten wird die Funktionalität maßgeblich durch Objekte zur Verfügung gestellt. So werden u.a. *Object Factories*[1], also standardisierte Objekte angeboten, denen die Erzeugung von neuen Objekten als Aufgabe übertragen wird. Allerdings ist es nicht notwendig, dass eine Komponente Klassen oder Objekte enthält. Statt dessen kann sie genauso ausschließlich auf klassischen Prozeduren oder Funktionen basieren. Selbst wenn nach außen, d.h. zum Benutzer einer Komponente hin, Objekte erscheinen, muss die Komponente nicht auf objektorientierter Technologie basieren. So ist z.B. das R/3-System in der 4GL-Sprache ABAP/4 implementiert. Die Funktionalität des R/3-Systems kann aber über objektorientierte Schnittstellen, sogenannte Wrapper[2] (siehe [Gamma et al. 1995]), also Hüllen, die dem System nach außen ein objektorientiertes Erscheinungsbild geben, aufgerufen werden.

---

[1] Object Factories werden z.B. zur Vereinheitlichung des NEW-Operators in objektorientierten Systemen genutzt, da sie – vorausgesetzt sie wurden von der gleichen abstrakten Klasse abgeleitet – austauschbar sind, während das Argument des NEW-Operators in den meisten objektorientierten Sprachen immer eine fest vorgegebene Klasse sein muß.

[2] Im SAP-Jargon heißen diese Wrapper *Business Application Programming Interfaces* (BAPIs, siehe [Appelrath und Ritter 1999]).

Weitere Unterschiede zwischen Objekten und Komponenten sind, dass objektorientierte Komponenten mehrere Klassen enthalten können, andererseits eine Klasse immer Bestandteil genau einer Komponente ist.

Klassen und Komponenten sind i.d.R. abhängig von anderen Klassen bzw. Komponenten. Eine Klasse kann natürlich von anderen, beispielsweise abstrakten Klassen abhängig sein, die nicht Bestandteil der gleichen Komponente sind. Genauso können Komponenten von anderen Komponenten – beispielsweise aufgrund von Klassenbeziehungen – abhängig sein[1]. Hier unterscheiden sich beide Konzepte wesentlich durch ihre Bindungen untereinander. Komponenten sollen eine geringe Kopplung (siehe Abschnitt 3.2.6) haben, damit sie „gute" Komponenten sind. Die Abhängigkeiten zwischen Objekten definieren sich im Wesentlichen aus funktionalen Gesichtspunkten.

3.2.5 Komponenten versus Module

Aus den vorhergehenden Diskussionen wird deutlich, dass der Begriff „Komponente" eher mit dem *Modulbegriff* als mit dem Klassen- bzw. Objektbegriff zu vergleichen ist, denn Module (oder *Packages* unter Java) decken die in den Abschnitten 3.2.1 und 3.2.4 geforderten Eigenschaften relativ gut ab. Allerdings sind klassische Modul- oder Package-Techniken i.d.R. nicht programmiersprachenunabhängig und bieten keine Möglichkeiten zur Parametrierung bzw. Konfiguration an, wie sie für Komponenten-Frameworks oft gefordert werden. So ist es z.B. sinnvoll, Komponenten durch einfache Parametrierungsmechanismen an die besonderen Anforderungen des Einsatzbereiches, z.B. Zeitzonen oder Sprachen, anzupassen, indem sie mit anderen Komponenten komponiert werden. Gemäß klassischer Modulkonzepte müssten die Komponenten für jedes Einsatzgebiet zuerst geändert (z.B. durch Setzen bestimmter Konstanten) und dann neu erzeugt/übersetzt werden. Aus diesem Grund bieten moderne Komponentenmodelle zusätzliche Mechanismen (Ressourcendateien, Serialisierung von Konfigurationsobjekten usw.) an, die die Definitionszeit einer Komponente um eine *Konfigurationszeit* (vgl. Abschnitt 2.2) verlängert.

3.2.6 Granularität der Wiederverwendung

Die Entwicklung möglichst kleiner wiederverwendbarer Komponenten ist zeitaufwendig und kostenintensiv. Andererseits sind große Komponenten i.d.R. nicht so flexibel, da sie schon auf einen konkreten Anwendungsbereich zugeschnitten sind. Der Grad der Wiederverwendbarkeit hängt somit indirekt von der Größe der Komponenten ab, wird aber zudem auch durch die

---

[1] Diese Abhängigkeit wird in Programmiersprachen wie C/C++ und Java i.d.R. durch Include- bzw. Import-Anweisungen definiert.

Konfigurationsmechanismen einer Komponente bestimmt. Konfigurationsmechanismen erlauben es, auch komplexe Komponenten an verschiedene Anwendungsbereiche anzupassen. Hier stellt sich dann die Frage, wieviel Aufwand dem Anwendungsarchitekten und dem Konfigurierer (siehe Abschnitt 1.2) für das Verstehen und die Konfiguration einer Komponente zugemutet werden kann. Die Regel „Maximizing reuse minimizes use“ veranschaulicht dieses Problem.

Betrachtet man beispielsweise das R/3-System der SAP als eine einzelne Komponente, so ist sie aufgrund ihrer Größe relativ schwer wiederzuverwenden. Jedoch bieten zahlreiche Parameter eine Vielzahl an Möglichkeiten zur Anpassung an verschiedene Anwendungsbereiche.

Für die Frage, wie groß Komponenten sein bzw. welche Aufgaben eines Anwendungsbereichs sie abdecken sollten, spielen aus formaler Sicht die Begriffe „Kohäsion“ und „Kopplung“ eine Rolle. *Kohäsion* bezeichnet den Grad der wechselseitigen Verbundenheit der Elemente (Klassen, Interfaces, Subkomponenten) innerhalb eines Moduls. Die Kohäsion sollte „hoch“[1] sein, so dass alle Elemente eines Moduls beispielsweise möglichst einer Aufgabe oder einem zusammenhängenden Aufgabenbereich dienen. Haben Elemente eines Moduls keinerlei Beziehungen – auch nicht transitiver Art, also indirekt über andere Elemente –, dann sollten sie verschiedenen Modulen zugeordnet werden.

Die *Kopplung* sagt aus, wie stark die Beziehungen zwischen einzelnen Modulen eines Anwendungssystems sind. Die Anzahl der Beziehungen sowie der Umfang des Informationsflusses zwischen den Modulen kann Aufschluss darüber geben, ob zwei potentielle Module nicht doch besser zu einem zusammengeschlossen werden sollten. Eine hohe Kohäsion ist also erwünscht, eine zu große Kopplung ist hingegen zu vermeiden. Ein weiteres wichtiges Kriterium bei der Gestaltung von Modulen ist das der *Lokalität*, das aussagt, dass jede Definition eines Elementes (Klasse, Interface, Komponente usw.) so lokal wie möglich und so global wie nötig deklariert werden sollte.

Die Anforderungen an die Modularisierung gelten auch für die komponentenbasierte Anwendungsentwicklung. Ein Problem der heute verbreiteten großen betrieblichen Anwendungssysteme wie R/3 und BaanERP ist, dass sie auf zentralen, monolithischen Datenbankmanagementsystem operieren und diese auch als Kommunikationsschnittstelle nutzen.

---

[1] Die ordinale bzw. kardinale Bewertung von Kohäsion und Kopplung ist nicht trivial, da es unterschiedliche Arten der Verbindung von Elementen einer Programmierumgebung gibt, die unterschiedlich stark gewichtet werden können. So unterscheidet man beispielsweise funktionale, sequentielle, zeitlich und andere Arten der Kohäsion, die für unterschiedliche Modulkonzepte unterschiedlich gewichtet werden müssen – je nach Zielsetzung der Modularisierung.

Selbst wenn sie in der darüber liegenden Anwendungsebene in Module gegliedert sind, wird dadurch die Forderung nach beliebiger Komposition von Komponenten kaum erfüllt.

Verbindliche Größenempfehlungen für Komponenten können nicht gegeben werden. In der Praxis bewegen sich die beiden Extreme aufeinander zu: Wurden in der Vergangenheit von Seiten der Forschung eher feiner granulare Komponenten und von Seiten der Hersteller betrieblicher Standardsoftwareprodukte eher monolithische Systeme verfolgt, scheint man sich nun aufeinander zu zu bewegen. Einen mittleren Ansatz zeigt beispielsweise die in der Praxis schon zu einer gewissen Bedeutung gelangte Spezifikation der OAGIS (Open Applications Group Integration Specification, siehe [OAG 1999]) auf.

3.2.7 White- und Black-Box-Abstraktion

Komponenten sind Elemente der Softwareentwicklung, die typischerweise in verschiedenen Anwendungsbereichen wiederverwendet werden sollen. Um die Wiederverwendung zu fördern, benötigt der Anwendungsarchitekt ein grundlegendes Verständnis über Schnittstellen und Verhalten einer Komponente. Die grundlegendste Art, dieses Verständnis aufzubauen, besteht darin, dass der Anwendungsarchitekt sich den Programmcode einer Komponente anschaut und daraus Schnittstellen und Verhalten ableitet. Er kann dann von der Implementierung abstrahieren und sich ein abstraktes – mentales – Modell dieser Komponente aufbauen. Bei dieser Art der Abstraktion, der sogenannten *White-Box-Abstraktion*, besteht das Problem, dass man auf Informationen zurückgreifen muss, die für ein Verständnis eigentlich gar nicht notwendig wären, deren Berücksichtigung jedoch beim Austausch durch eine Komponente eines anderen Herstellers zu Kompatibilitätsproblemen führen kann.

Wesentlich flexibler als diese sogenannte White- ist die *Black-Box-Abstraktion*, bei der dem Anwendungsarchitekten nur Informationen über die Schnittstellen und die Spezifikation des Verhaltens mitgeteilt werden, Implementierungsdetails jedoch verborgen bleiben. Eine Black-Box-Abstraktion enthält nur die Informationen, die dem Architekten zum Einbau der Komponente unbedingt zur Verfügung stehen müssen. Die Schnittstellen definieren die Aufrufmöglichkeiten, und eine Verhaltensspezifikation definiert die möglichen Reihenfolgen der Schnittstellenaufrufe einer Komponente.

Weiterhin kann man noch *Grey-Box-Abstraktionen* unterscheiden, die einen Mittelweg darstellen und einen kontrollierten Teil ihrer Implementierung nach außen preisgeben. Eine solche „partielle Implementierung“ kann als Verfeinerung (Refinement, siehe z.B. [Back und Wright. 1998]) der Spezifikation in Richtung auf die Implementierung verstanden werden.

In dieser Arbeit wird ausschließlich die Black-Box-Abstraktion zugrunde gelegt.

3.2.8 Scripting

Unter *Scripting* wird die Verbindung (*Glueing*) von Komponenten mittels einer speziellen, meist imperativen Programmiersprache verstanden. Scripting ist mitunter notwendig, weil Komponenten nicht immer direkt im Sinne eines „Plug-in and Play“, d.h. ohne zusätzliche Programmierung, miteinander gekoppelt werden können, obwohl sie inhaltlich durchaus zueinander passen würden. Statt dessen ist oft noch Anpassungsarbeit notwendig, die durch Scripting abgedeckt wird. Programmiersprachen für das Scripting sind i.d.R. weniger mächtig als normale Programmiersprachen, dafür aber einfacher zu handhaben. Die speziellen Programme, die im Rahmen der Entwicklung von Skripten entstehen, fügen dem bestehenden System nur neues Verhalten, aber keine neuen persistenten Zustände hinzu. Ein Skript ist letztlich also ein „normales“ Programm und kann somit, wenn es bestimmte Konventionen einhält, auch als Komponente (script components, siehe [Nierstrasz, Gibbs und Tsichritzis 1992]) verstanden werden. Aus diesem Grund wird in dieser Arbeit auf eine zusätzliche Berücksichtigung des Scriptings verzichtet. Für den Fall, dass zwei Komponenten technisch oder inhaltlich nicht zueinander passen, muss eine zusätzliche Komponente entwickelt werden, die eine Kopplung dieser Komponenten ermöglicht.

Da in dieser Arbeit die Entwicklung von Komponenten keine Rolle spielt, wird durch den Ausschluss des Scriptings das Problem der Entwicklung betrieblicher Anwendungssysteme auf die Parametrierung und Konfiguration der Komponenten reduziert.

3.2.9 Komponenten, Konfigurationen und Instanzen

Komponenten werden zur Definitionszeit zu größeren Komponenten zusammengesetzt. In bestimmten Situationen kann es sinnvoll sein, die Verbindungen, die dazu zwischen den Komponenten konfiguriert werden müssen, in ihrer Anzahl flexibel zu gestalten. Dafür führe man sich folgende Situation vor Augen: Eine Komponente (genauer: ein Komponenten-Framework) C kann je nach Einsatzbereich entweder mit einer Komponente X_1 oder mit einer Komponente X_2 zusammenarbeiten. Wenn nun eine Komponente A die Komponente C gleichzeitig sowohl mit der Teilkomponente X_1 als auch mit der Komponente X_2 (siehe Abbildung 3.2, links) betreiben möchte, muss die Komponente C zweimal konfiguriert werden: einmal mit der Teilkomponente X_1 und einmal mit X_2. Um diesen Sachverhalt ohne eine Neuübersetzung der Komponente – zur Erinnerung: eine Komponente liegt immer in binärer Form vor – realisieren zu können, müssen die Konfigurationsinformationen außerhalb der Komponenten gespeichert werden, z.B. in einem vom System-Framework (siehe Abschnitt 1.2) zur Verfügung gestellten standardisierten Naming-and-directory-Service oder direkt in einer Ressourcen-Datei zur Komponente, deren Inhalte zur Laufzeit gelesen und interpretiert wird.

Wenn ein Konfigurationsmechanismus erlaubt, eine Komponente C mehrmals (hier als C‘ und C‘‘, siehe Abbildung 3.2, rechts) zu konfigurieren, stellt sich die Frage, was nun tatsächlich eine Komponente ist. Ist C eine Komponente, sind die beiden Konfigurationen C‘ und C‘‘ Komponenten, oder ist C inklusive der Konfigurationen eine Komponente? In dieser Arbeit werden alle drei Elemente als Komponenten bezeichnet. X_1, X_2 und C sind atomare Komponenten. C ist zusätzlich parametrierbar und deswegen ein Komponenten-Framework. C‘, C‘‘ und A sind komplexe Komponenten, die durch Zusammensetzen von C und X_1, C und X_2 sowie C‘ und C‘‘ entstanden sind.

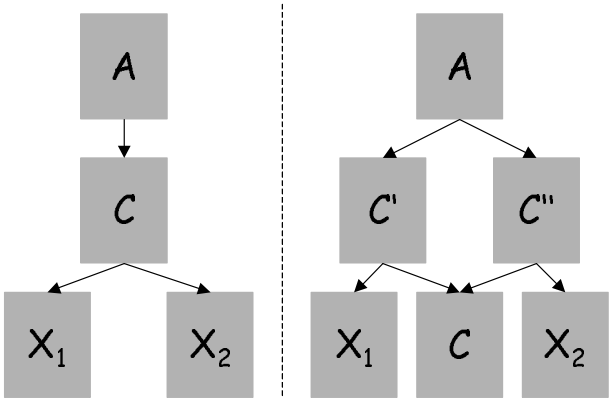


Abbildung 3.2: Komponenten vs. Komponenteninstanzen

In der Praxis ist die Terminologie diesbezüglich nicht eindeutig. So werden beispielsweise im JavaBeans-Komponentenmodell (siehe [SUN 1997]) u.a. auch GUI-Elemente (also konkrete Objekte) wie Buttons als Komponenten (JavaBeans) betrachtet. Diese Komponenten können mit anderen Komponenten zu größeren Komponenten zusammengesetzt werden. Um eine solche Konfiguration zu sichern, kann sie durch einen Persistenzmechanismus (Serialisierung, siehe [Flanagan 1997]) abgespeichert werden. Eine JavaBeans-Komponente besteht dann u.a. aus den benötigten Klassendefinitionen („class“-Dateien) und den serialisierten Objekten (Konfigurationsinformationen). Dieses Prinzip widerspricht dem bislang hier präsentierten Verständnis von Komponenten, dem u.a. die Forderung zugrunde liegt, dass eine Komponente keinen persistenten Zustand besitzt. Nach der in Abschnitt 3.2.1 präsentierten Definition und den in Abschnitt 3.2.4 aufgezählten Eigenschaften von Komponenten dürfte eine Java-Komponente dann nur aus den Klassendefinitionen bestehen. JavaBeans sind in diesem Sinne dann Instanzen ihrer Komponenten (*Komponenteninstanzen*) und keine Komponenten.

Etwas anders sieht es im Enterprise-JavaBeans-Komponentenmodell (siehe [Matena und Hapner 1999]) aus. Dort werden die Konfigurationsinformationen zu einer Komponente

(Enterprise Bean[1]) in einer XML-Datei – dem sogenannten *Deployment Descriptor* – abgelegt. Aus diesen Informationen werden zur Laufzeit Objekte erzeugt, die das Laufzeitverhalten der Komponente steuern. U.a. entstehen sogenannte EJBHome-Objekte, die den zentralen Zugriffspunkt auf die Funktionalität der Komponente – im Wesentlichen Methoden zur Verwaltung von Geschäftsobjekten – zur Verfügung stellen. Allerdings kann es mehrere EJBHome-Objekte, also mehrere Instanzen zu einer Komponente geben. Im EJB-Umfeld werden begrifflich verwirrend gerade diese EJBHome-Objekte als Komponenten bezeichnet. Tatsächlich handelt es sich jedoch immer noch um ein und dieselbe Komponente, die diesen Instanzen zugrunde liegt.

In dieser Arbeit wird also an der Definition (siehe Abschnitt 3.2.1) festgehalten, dass zu einer Komponente nicht die Instanzen einer Komponente zählen. Allerdings können zu einer Komponente und insbesondere zu einem Komponenten-Framework Konfigurationsmechanismen gehören, die der Parametrierung dienen. So ist eine Enterprise Bean inklusive ihrer Konfigurationsmechanismen eine Komponente[2]. Wenn diese Komponente nun zweimal eingesetzt wird, also zwei Deployment Descriptoren besitzt, so handelt es sich immer noch um eine Komponente, die zur Laufzeit zwei Instanzen besitzt, die sogar über Rechnersysteme hinweg verteilt sein können.

3.3 Komponentenschnittstellen

3.3.1 Direkte versus indirekte Schnittstellen

Grundsätzlich kann man Komponentenschnittstellen in direkte Schnittstellen und indirekte Schnittstellen unterteilen. *Direkte Schnittstellen* entsprechen prozeduralen Aufrufen, wie man sie aus klassischen Softwarebibliotheken kennt. *Indirekte Schnittstellen* sind Schnittstellen von Objekten, die durch Komponenten zur Verfügung gestellt werden. In den meisten objektorientierten Programmiersprachen werden direkte Schnittstellen durch statische Methoden (Klassenmethoden) zur Verfügung gestellt.

---

[1] Das Komponentenmodell von SUN heißt Java EnterpriseBeans. Eine einzelne Komponente heißt Enterprise Bean.

[2] Diese Konfigurationsmechanismen können auch auf Basis von Objekten realisiert sein, deren Klassen Bestandteil der Komponente sind. Jedoch müssen diese Objekte klar als Konfigurationsobjekte zu erkennen sein. Dies ist im JavaBeans-Komponentenmodell nicht der Fall. Hier werden alle Objekte transitiv serialisiert, wodurch die Abgrenzung zwischen Konfigurations- und Laufzeitobjekten nicht mehr gegeben ist. Ein besonderes Problem in diesem Modell bringt die Versionierung mit sich. Möchte man z.B. fehlerhafte Klassendefinitionen gegen neue austauschen, so müssen u.U. auch die Konfigurationsinformationen neu erzeugt werden. Dies ist eine mittlerweile bekannte Problematik beim JavaBeans-Komponentenmodell.

Nachteil einer indirekten Schnittstelle ist, dass der Nutzer einer Komponente auch Schnittstellen unbekannter Komponenten aufrufen kann, sobald er Referenzen auf Objekte dieser Komponenten erhält. Für den Benutzer ist somit der Verantwortungsbereich einer Komponente nicht mehr nachvollziehbar, da er die Schnittstellen der Objekte nicht eindeutig den Komponenten zuordnen kann. Bei der Wiederverwendung von Komponenten mit direkten Schnittstellen reicht es aus, die Version der aufzurufenden Schnittstelle zu prüfen. Bei indirekten Schnittstellen kann es vorkommen, dass Komponenten sich gegenseitig Objekte mit inkompatiblen Schnittstellenversionen vermitteln. Dies kann, wenn nicht jede Übergabestelle für Objektreferenzen geprüft wird, zu unvorhersehbaren Problemen führen.

3.3.2 Schnittstellen und Verträge

Eine geeignete Metapher für die Vereinbarung von Schnittstellen und Verhalten[1] sind „Verträge“ (Kontrakte), die zwischen Komponenten und Frameworks abgeschlossen werden und damit bei der Entwicklung komplexer Komponenten und von Anwendungssoftwareprodukten berücksichtigt werden müssen. Der Vertrag definiert, welche Informationen einer Komponente beim Aufruf zugesichert bzw. welche Ergebnisse nach dem Aufruf von der aufrufenden Komponente benötigt werden. So wird erreicht, dass Komponentenhersteller unabhängig voneinander Komponenten herstellen können, die später durch den Anwendungsarchitekten im Idealfall „problemlos“ zusammengesteckt werden können.

Es gibt verschiedene Sprachmittel, um diese Verträge zu formulieren. Dabei geht es nicht nur um die Festlegung der Aufrufschnittstellen von Prozeduren bzw. Methoden, sondern auch um die Zusicherung von bestimmten Bedingungen, die der Anwendungsarchitekt vor dem Aufruf und die die Komponente nach dem Aufruf zusichert. Solche Bedingen können beispielsweise in sogenannten Vor-/Nachbedingungspaaren (*Pre-/Post-Conditions*) formuliert werden. Diese Bedingungspaare können mehr oder weniger formalisiert sein. So findet man auf dem Gebiet der betrieblichen Anwendungssysteme vornehmlich natürlichsprachliche Formulierungen solcher Verträge, während in sicherheitskritischen Bereichen zunehmend mathematisch fundierte Methoden genutzt werden (siehe [Olderog und Steffen 1999]).

Ein Problem der Formulierung der Verträge mittels Pre-/Post-Conditions ist, dass das Aufrufverhalten einer Prozedur bzw. Methode i.d.R. nur isoliert betrachtet wird. Dies reicht bei einfachen funktionalen Bibliotheken vollkommen aus. In Systemen mit mehreren

---

[1] Im Kontext dieser Arbeit werden vornehmlich verhaltensbezogene Aspekte von Anwendungssystemen betrachtet. Die in diesem Abschnitt formulierten Verträge ließen sich natürlich auch auf nicht-funktionale Anforderungen, wie Performanz, Ressourcenverbrauch oder Benutzungsfreundlichkeit erweitern.

Prozessen werden jedoch oft asynchrone Publish-/Subscribe- oder Callback-Mechanismen (siehe Abschnitt 3.2.2) zur Verfügung gestellt, die der Benachrichtigung gekoppelter Komponenten dienen. So werden typischerweise Client-Komponenten über Zustandsänderungen von Objekten einer Server-Komponente mittels solcher Mechanismen benachrichtigt. Zustandsinformationen können jedoch nicht so einfach innerhalb der Pre-/Post-Conditions formuliert werden. Formal wäre dies zwar prinzipiell möglich, jedoch wird davon aus Komplexitätsgründen meist kein Gebrauch gemacht.

Durch Nichtbeachtung dieser Zustände können Komponentensysteme in unvorhersehbare Systemzustände überführt werden, z.B. durch den unkontrolliert parallelen Aufruf mehrerer Methoden eines Objektes (*Re-entrance*). Neben den Pre-/Post-Conditions gibt es deshalb eine Reihe weiterer Sprachen zur Formulierung der Verträge, die gültige Reihenfolgen auch asynchroner Interaktionen und dadurch insbesondere den Zustand einer Komponente berücksichtigen. Ein Beispiel ist durch die Spezifikation der Interaktions- bzw. Kommunikationssequenzen gegeben. Kommunikationssequenzen können auf Basis verschiedener Sprachansätze (siehe Abschnitt 7.4) beschrieben werden.

3.3.3 Ereignisse und Nachrichten

Komponenten benötigen i.d.R. andere Komponenten, um arbeiten zu können. Aus diesem Grund werden Komponenten untereinander verbunden. Die herkömmlichen Prozedur-/Methodenaufrufe, die diesem Zweck dienen, basieren auf einem *Pull-Modell*. Im Pull-Modell ruft eine Komponente die Prozedur einer anderen Komponente auf, um Informationen zu ermitteln („herauszuziehen“). Andererseits kann es auch notwendig sein, dass eine Komponente immer dann informiert werden möchte – d.h., dass eine Prozedur/Methode dieser Komponente aufgerufen werden soll –, wenn ein bestimmtes Ereignis (Event) bevorsteht, gerade stattfindet oder abgeschlossen wurde. Diese Art von Verbindung wird als *Push-Modell* bezeichnet.

Während im Pull-Modell direkte Verbindungen zwischen Komponenten ausreichen, werden für das Push-Modell meist komplexere Verbindungsstrukturen benötigt. Dazu werden spezielle Dienste – sogenannte Event-Dienste – entwickelt, die z.B. Ereignisse aufnehmen und weiter vermitteln. Denkbar ist auch, dass die Kommunikation nicht auf dem Aufrufen bzw. dem Aufgerufen-Werden von Prozeduren/Methoden basiert, sondern ausschließlich auf einem Versenden von Nachrichten (Messages). Auch dafür gibt es entsprechend standardisierte Dienste, sogenannte Message-Dienste[1]. Bei beiden Verfahren wird die

[1] Die Unterscheidung von Ereignissen und Nachrichten (Events und Messages) ist in der Fachliteratur nicht eindeutig. Der Hauptunterschied liegt im Wesentlichen darin, dass bei einem Ereignis direkt eine Methode/Prozedur mit den relevanten Informationen aufgerufen wird. Beim Verschicken von

Konfiguration der Verbindungen nicht mehr notwendigerweise im Programmcode (Definitionszeit) vorgegeben, sondern kann dadurch vollzogen werden, dass eine an bestimmten Informationen interessierte Komponente sich zur Laufzeit bei einer anderen Komponente anmeldet (*Publish/Subscribe-Verfahren*).

Im Push-Modell kann es sinnvoll sein, Vermittlungsdienste einzurichten, die Nachrichten oder Ereignisse aufnehmen und an interessierte Empfänger weitergeben, da dann nicht jede empfangende Komponente mit jeder sendenden Komponente direkt verbunden werden muss. Vermittlungsdienste sind ebenfalls Komponenten, die auch ein- und ausgehende Schnittstellen besitzen. In Abbildung 3.3 wird z.B. ein Server S über eine Vermittlungsstelle V an zwei Clients C1 und C2 angebunden. Eine Nachricht von S wird dann über V sowohl an C1 als auch an C2 weitergeleitet. In den Vermittlungskomponenten – oft wird auch von Empfängergruppen oder von Kanälen (Channels) gesprochen – können nun „intelligente“ Filter eingesetzt werden, die über differenzierte Kriterien die Nachrichten zuordnen. Ähnlich wie beim Scripting (siehe Abschnitt 3.2.8) sind das Event- und Messagehandling Aufgaben, die letztlich durch Komponenten übernommen werden. Dies bedeutet für das in dieser Arbeit zu entwerfende Komponentenmodell, dass keine besonderen Aspekte zur Nachrichten- oder Ereignisverarbeitung berücksichtigt werden müssen. Sie werden, wie alle anderen Aspekte auch in Komponenten gekapselt.

3.3.4 Eingehende und ausgehende Schnittstellen

Die Schnittstellen einer Komponente können in ein- und ausgehende Schnittstellen (siehe Abbildung 3.3) unterschieden werden. Die Richtung sagt jedoch nichts über den tatsächlichen Informationsfluss aus. So können Schnittstellen beider Richtungen Informationen aufnehmen und abgegeben, da ja Methoden und Prozeduren übergebene Parameter abändern und ergänzen können sowie u.U., wie oft üblich, Funktionswerte (*Return-Werte*) zurückgeben.

Nachrichten wird dazu erst ein spezielles Nachrichtenobjekt (Message) erzeugt, das die zu verschickenden Informationen enthält. Erst dann werden entsprechende Methoden/Prozeduren aufgerufen, denen dann die Informationen übergeben werden. Nachrichten haben damit auch den Vorteil, dass sie persistent gespeichert werden können.

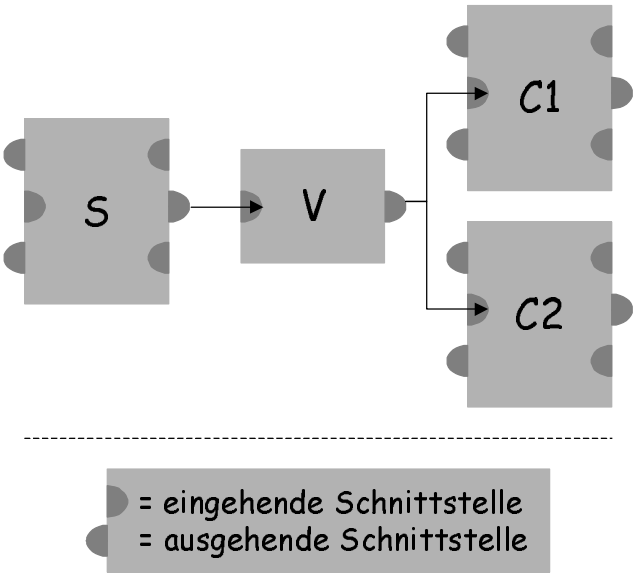


Abbildung 3.3: Vermittlung zwischen sendenden und empfangenden Komponenten

Die Verbindungen zwischen Komponenten sind unabhängig von Import oder Export von Schnittstellen, wie sie in Paketen (Packages) oder Modulen definiert werden. Natürlich müssen Komponenten, die direkt miteinander kommunizieren wollen, gleiche Schnittstellen unterstützen. Jedoch ist noch nichts darüber ausgesagt, welche Komponente tatsächlich später mit welcher anderen Komponente zusammenarbeitet.

In Abbildung 3.4 sind zwei Komponenten K1 und K2 zu sehen. K2 muss K1 importieren, weil sie durch A das Interface A‘ implementiert und durch D auf Objekte der Klasse C verweist. Die Aufrufbeziehungen können trotz der unidirektionalen Importbeziehung in beide Richtungen weisen: K2 kann von ihren Objekten (z.B. der Klasse D) über Verweise Objekte von K1 (z.B. der Klasse C) aufrufen – dies ist eine sogenannte *Request/Reply-Beziehung*. K1 kann von ihren Objekten (z.B. der Klasse B) Objekte von K2 (z.B. der Klasse A) aufrufen – dieses ist eine *Callback-Beziehung*.

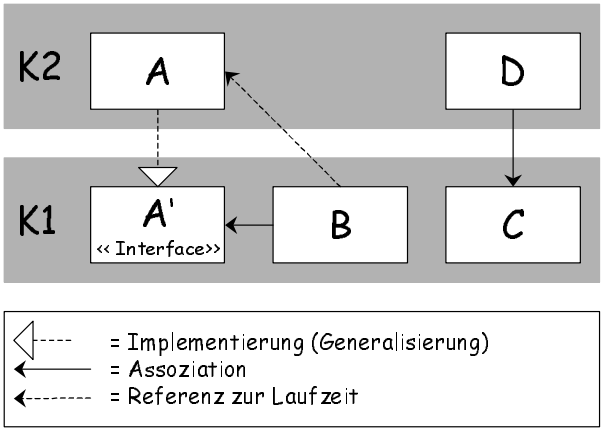


Abbildung 3.4: Abhängigkeiten zwischen Komponenten (1)

In dieser hierarchischen, azyklischen Schichtenarchitektur kann die Komponente K2 einfach gegen eine andere Komponente – die dann ebenfalls K1 importieren muss – ausgetauscht werden. Soll allerdings die Komponente K1 unter Beibehaltung von K2 ausgetauscht werden, ist dies in den meisten Programmiersprachen so nicht möglich, da die Importbeziehungen „fest verdrahtet“ sind. Um K1 austauschen zu können, muss deshalb eine weitere „Komponente“ K0 (siehe Abbildung 3.4) definiert werden, in die die Schnittstellenvereinbarungen für die Kommunikation zwischen K1 und K2 ausgelagert werden. Nun müssen K1 und K2 diese Vereinbarungen bei der Implementierung berücksichtigen und stehen somit gleichberechtigt auf einer Ebene nebeneinander. Beide Komponenten können nun ohne Probleme gegen Ersatzkomponenten ausgetauscht werden – vorausgesetzt sie importieren K0 und sind ansonsten substituierbar (siehe Abschnitt 4.1.2).

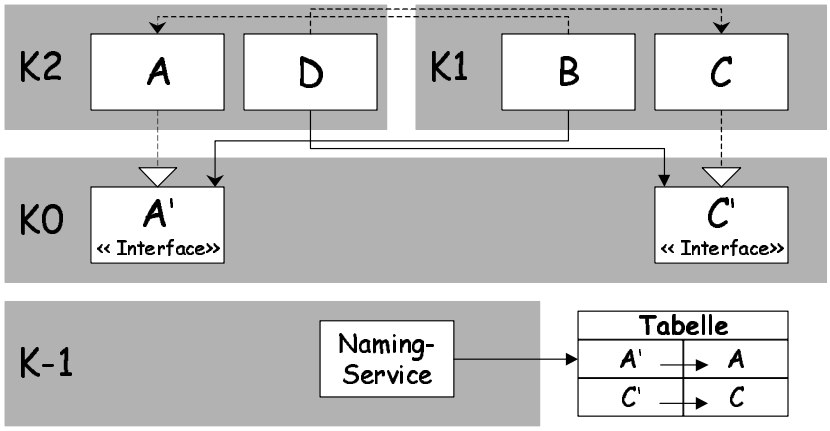


Abbildung 3.5: Abhängigkeiten zwischen Komponenten (2)

Eine Frage bleibt jedoch offen: Wie kommen die Referenzen zwischen den Objekten ein und derselben Ebene – im Beispiel zwischen den Klassen B und A sowie D und C – zur Laufzeit zustande, wenn die Komponenten K1 und K2 sich nicht gegenseitig importieren dürfen und somit auch nicht direkt Objekte der Klassen erzeugen können? Um dieses Problem zu lösen, wird im Beispiel in Abbildung 3.5 in einer eigenständigen tieferen Ebene K-1 (System-Framework) ein sogenannter *Naming-Service* zur Verfügung gestellt, der Funktionalitäten der darüber liegenden Ebenen einander vermittelt. Dieser Service greift auf eine Tabelle zurück, die zur Konfigurationszeit gefüllt wird und verschiedenen standardisierten Bezeichnern (im Beispiel möglicherweise die Namen der Interfaces in Ebene K0 (A‘ oder C‘)) konkrete Klassen (im Beispiel die Klassen A und C) zuordnet. Möchte z.B. die Komponente K1 eine Referenz von einem Objekt der Klasse B auf ein neues Objekt der Klasse A der Komponente K2 erzeugen, so ermittelt sie über den Naming-Service den zum Interface A‘ assoziierten Klassennamen A und erzeugt dann ein Objekt dieser Klasse. Möchte man nicht nur Objekte einer Klasse erzeugen, sondern auch bestehende Objekte finden, werden den Token (hier: A‘ und C‘) nicht Klassen, sondern komplexere Objekte der jeweiligen Komponenten – sogenannte Factories (siehe Abschnitt 3.2.4) – zugeordnet, über die diese Dienste erbracht

werden können. Von dieser Idee, die auf dem *Abstract-Factory-Muster* (siehe [Gamma et al. 1995]) basiert, wird u.a. im San Francisco Framework der Firma IBM (siehe [Abinava et al. 1998]) Gebrauch gemacht.

K0 ist keine Komponente i.e.S., da sie ohne zusätzliche Komponenten nicht einsetzbar ist. Komponenten wie K0 bezeichnet man als *Komponenten-Frameworks*, da sie Slots (Hot Spots) zur Erweiterung durch weitere Komponenten (hier: K1 und K2) bieten.

Für die Konfiguration eines komponentenbasierten Anwendungssystems sind die Importbeziehungen notwendigerweise fest vorgegeben[1].

Im Beispiel der Abbildung 3.5 wird davon ausgegangen, dass der Naming-Service Zuordnungen auf Ebene der Interfaces und Klassen und nicht auf Ebene von Komponenten vornimmt. Die Praxis hat jedoch gezeigt, dass dieses Vorgehen zu feingranular ist und nicht den Anforderungen eines *Markts für Komponenten* (vgl. Abschnitt 3.2.4) gerecht wird. Aus diesem Grund geht der Trend dahin, Naming-Services für Komponenten zu nutzen. Beispiele dafür sind die Enterprise JavaBeans (EJB) der Firma SUN (siehe [Matena und Hapner 1999]). Im EJB-Komponentenmodell wird pro Komponente ein ausgezeichnetes Objekt – dieses Objekt muss das Interface EJBHome implementieren – definiert, dass an einen zentralen Naming-Service angemeldet werden muss. Über dieses Objekt können Objekte der Komponente erzeugt, gesucht und gelöscht sowie spezielle Geschäftstransaktionen ausgeführt werden.

3.4 Softwarekonfigurationsmanagement

Zur Verwaltung von Softwarekomponenten werden spezielle Methoden und Werkzeuge benötigt. Neben der Verwaltung von Versionen, die im Folgenden behandelt wird, werden auch Funktionen zur Strukturdarstellung größerer Softwareprodukte und zunehmend zur Koordination von Entwicklungs- und Konfigurationsteams benötigt.

Zu den Aufgaben der Komponentenverwaltung gehört es, verschiedene *Versionen* einer Komponente identifizierbar und zugreifbar zu machen, zu klassifizieren und zu speichern. Die Versionen können in Revisionen und Varianten unterschieden werden. Eine *Revision* soll zeitlich vorhergehende Versionen einer Komponente ersetzen. Eine neue Revision wird i. d. R. durch Modifikation der letzten Revision erzeugt. Demgegenüber ist eine *Variante* eine Alternative zu einer bestehenden Komponente. Verschiedene Varianten können parallel weiterentwickelt werden. Die Struktur des Systems, d.h. das Systemmodell, die Beziehungen

[1] Es gibt allerdings in einigen „exotischen“ Programmiersprachen wie z.B. Objective-C (siehe [Pinson und Wiener 1991]) die Möglichkeit, Generalisierungsbeziehungen erst zur Laufzeit zuzuordnen (dynamic inheritance).

zwischen den Komponenten und die Konsistenzbedingungen müssen verwaltet und bei der Konfiguration eingehalten werden. Teamfunktionen ermöglichen dem Entwicklungsteam, das Produkt koordiniert zu entwickeln und zu warten. Dazu gehören z.B. Funktionen zur Konfliktauflösung von zwei Versionen einer Komponente, die wieder in eine, beide Zweige vereinigende Version zusammengeführt werden sollen.. Diese Funktionalität wird von verschiedenen Konfigurationsmanagementmodellen mit unterschiedlichen Schwerpunkten behandelt. In [Feiler 1991] werden die bekanntesten Modelle vorgestellt: das Checkout/Checkin-Modell, das Änderungsmengenmodell, das Kompositionsmodell und das Modell der langen Transaktionen.

3.4.1 Das Checkout/Checkin-Modell

Das älteste Softwarekonfigurationsmodell ist das *Checkout/Checkin-Modell*. In diesem Modell werden alle Komponentenversionen in einem zentralen Repository verwaltet. Auf eine Version einer Komponente wird zugegriffen, indem sie aus dem Repository in ein privates Verzeichnis des Nutzers (Programmierers) kopiert (check out) und nach der Bearbeitung wieder in das Repository zurückkopiert wird (check in). Eine typische Implementierung dieses Systems ist Tichys *Revision Control System* (RCS, siehe [Tichy 1985]). Das Checkout/Checkin-Modell unterstützt vornehmlich die Verwaltung von Varianten und Revisionen einzelner Komponenten, aber nicht von Konfigurationen mehrerer Komponenten.

3.4.2 Das Änderungsmengenmodell

Im Änderungsmengenmodell besteht die grundlegende Idee darin, anstatt neuer Versionen nur Änderungen bezogen auf einen Anfangszustand zu speichern. Inhaltlich zusammengehörende Änderungen an verschiedenen Komponenten können dann in Änderungsmengen (Patches) zusammengefasst werden. Änderungen brauchen sich nicht auf eine bestimmte Version einer Komponente beziehen. Statt dessen wird für eine zu erzeugende Version festgelegt, welche Änderungen einbezogen werden sollen. Problematisch ist dabei die Darstellung der Abhängigkeiten zwischen den Änderungen: Das System muss sicherstellen, dass bei Anwendung einer Änderungsmenge C auf eine Komponente bereits alle Änderungen C‘, auf denen C basiert, durchgeführt worden sind. Insgesamt stellt dieses Modell durch die Ausrichtung an den Änderungen als eigenständige Objekte ein am Zeitverlauf der Softwareentwicklung (Revisionen) orientiertes Modell dar. Wie im Checkout/Checkin-Modell steht auch hier die Verwaltung einzelner Komponenten im Vordergrund. Strukturelle Beziehungen zwischen den Komponenten können allenfalls am Rande durch die Abhängigkeiten zwischen Änderungsmengen ausgedrückt werden.

3.4.3 Das Kompositionsmodell

Im Kompositionsmodell wird die Softwarekonfiguration von der Komponenten- auf die Systemebene erweitert. Die Struktur des Systems wird in einem *Systemmodell* definiert und

die Komponentenversionen durch *Auswahlregeln* bestimmt. Eine konkrete Konfiguration, d.h. eine Menge ausgewählter Komponentenversionen, muss bestimmten *Konsistenzbedingungen* genügen. Die Vorgehensweise in diesem Modell ist dann wie folgt: Zunächst wird das System auf Basis seiner Komponenten beschrieben. Dann wird in einem eigenen Schritt für jede Komponente eine geeignete Version ausgewählt. Aus einfachen Komponenten können komplexere Komponenten zusammengesetzt (komponiert) werden. Zur Repräsentation des Systemmodells werden Relationen zwischen den einfachen Komponenten und den komplexen Komponenten genutzt. Das Systemmodell kann beispielsweise durch einen *UND/ODER-Graphen*[1] visualisiert werden. Die Aggregationsbeziehungen zwischen komplexen Komponenten und ihren konstituierenden, einfacheren Komponenten werden durch UND-Kanten dargestellt. Eine UND-Kante von einer Komponente E zu einer komplexen Komponente K besagt dann, dass E eine Teilkomponente von K ist. Alternative Komponenten können durch die Zwischenschaltung von ODER-Knoten repräsentiert werden Ein ODER-Knoten ist eine abstrakte Komponente, die verschiedene alternative Realisierungen verkörpert. Eine Kante von einer Komponente K zu einem ODER-Knoten O bedeutet, dass K eine Version (Variante) von O ist. Eine Kante von einer abstrakten Komponente (ODER-Knoten) O zu einer komplexen Komponente K besagt, dass O Teilkomponente von K ist.

In Abbildung 3.6 sieht man einen UND/ODER-Graphen, der eine komplexe Komponente X beschreibt, die aus drei einfacheren Komponenten zusammengesetzt wird. Die Komponente B ist fest vorgegeben. Für die beiden anderen Komponenten stehen jeweils zwei Alternativen zur Verfügung. Insgesamt gibt es also vier mögliche Konfigurationen, nämlich {A1, B, C1}, {A1, B, C2}, {A2, B, C1} und {A2, B, C2}.

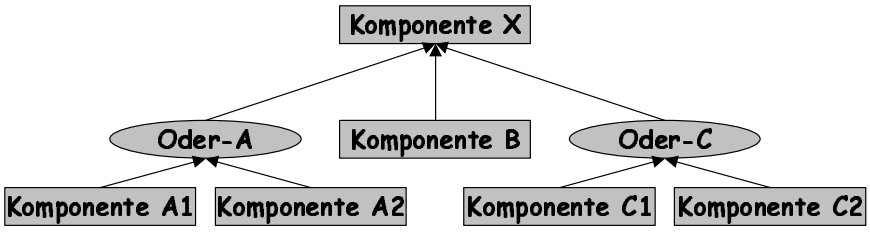


Abbildung 3.6: UND/ODER-Graph (1)

Es ist leicht nachvollziehbar, dass größere Konfigurationsprobleme nicht so einfach durch reine UND/ODER-Graphen darstellbar sind. Existieren beispielsweise Abhängigkeiten zwischen den Alternativen mehrerer abstrakter Komponenten (ODER-Knoten), so werden die UND/ODER-Graphen recht schnell unüberschaubar. Dürften im Beispiel aus Abbildung 3.6

[1] In der Konfigurationsliteratur (z.B. [Bach 1994]) wird anstatt von einem UND/ODER-Graph oft von einem UND/ODER-Baum gesprochen, da Komponenten(typen), von denen mehrere Exemplare (Instanzen) eingesetzt werden, mehrfach als eigenständige Blätter dargestellt werden.

nur Versionen eingesetzt werden, bei denen die Nummer der Komponente aus dem Zweig Oder-C größer oder gleich der Nummer der Komponenten des Zweigs Oder-A ist, so wären die Konfigurationen {A1, B, C1}, {A1, B, C2} und {A2, B, C2} gültig.

Diese drei Alternativen ließen sich als Graph wie in Abbildung 3.7 darstellen. Um solche komplexen und pflegeintensiven Darstellungen zu vermeiden, werden in datenbank- und wissensbasierten Ansätzen weitere Relationen – sogenannte *Constraints* (*Konsistenzbedingungen*) – definiert, mittels derer komplexere Sachverhalte ausgedrückt werden können. In das Beispiel aus Abbildung 3.6 könnte man z.B. einen Constraint zwischen den ODER-Knoten Oder-A und Oder-C einführen, der gerade die oben beschriebene Beziehung fordert.

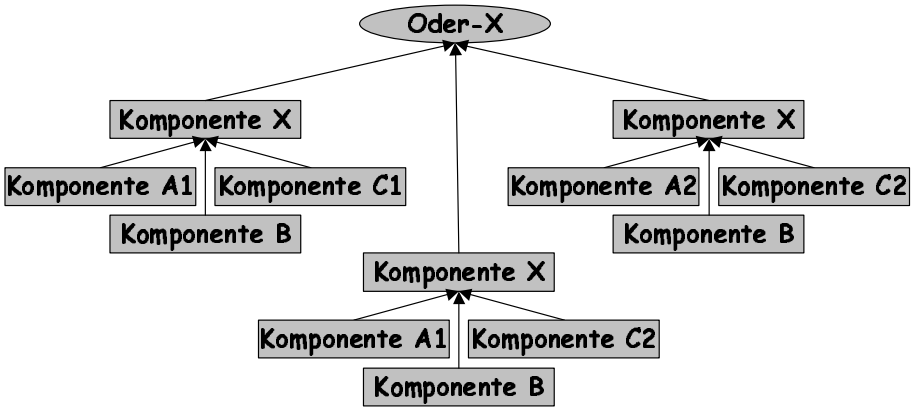


Abbildung 3.7: UND/ODER-Graph (2)

3.4.4 Das Modell der langen Transaktionen

Ein hier bisher noch nicht betrachteter Aspekt der Modelle zur Softwarekonfiguration ist die Teamfähigkeit. Software wird i.d.R. in einem Team entwickelt, dessen Mitglieder parallel arbeiten. Die Aufgabe eines Werkzeuges zum Softwarekonfigurationsmanagement besteht in einem solchen Umfeld zunächst darin, die individuellen Änderungen der einzelnen Softwareentwickler voneinander zu isolieren und – wenn ein Arbeitsschritt abgeschlossen ist – die Änderungen im System zu verbreiten. Das zentrale Konzept zur Erfüllung dieser Aufgabe ist das Modell der langen Transaktionen, auch Arbeitsbereiche (Workspaces) genannt. Wenn ein Softwareentwickler eine lange Transaktion startet, isoliert er die anderen Teammitglieder von seinen Änderungen. Aus seiner Sicht hat er dann einen individuellen Arbeitsbereich. Auf diesen Arbeitsbereich kann z. B. wie auf ein virtuelles Dateisystem zugegriffen werden, so dass die Integration in eine Softwareentwicklungsumgebung leicht fällt. Die Veröffentlichung der Änderungen über die Grenze des Arbeitsbereiches hinaus ist eine explizite Operation des Werkzeuges zum Softwarekonfigurationsmanagement. Dafür muss es über eine Strategie zur Konfliktauflösung verfügen oder generell Konflikte mit einer Kooperationsstrategie durch explizites Sperren von Objekten (wie z. B. bei RCS) verhindern.

3.4.5 Auswahl eines Konfigurationsmodells für Softwarekomponenten

In dieser Arbeit steht die Komposition von betrieblichen Anwendungssoftwareprodukten und ihre Konfiguration zu konkreten Anwendungssystemen im Vordergrund. Offensichtlich ist dazu das Kompositionsmodell (siehe Abschnitt 3.4.3) besonders geeignet. Das Checkin/Checkout-Modell und das Änderungsmengenmodell betonen in besonderem Maße die zeitlichen Revisionen einzelner Komponenten, die in dieser Arbeit nicht berücksichtigt werden. Ebenso wird in dieser Arbeit die Teamfähigkeit des zugrunde liegenden Modells (vgl. Abschnitt 3.4) vernachlässigt.

Es sei allerdings an dieser Stelle betont, dass diese beiden Aspekte – Revisionen und Teamfähigkeit – auch im Zusammenhang der Konfiguration betrieblicher Anwendungssysteme eine wichtige Rolle spielen. So führen ständige Neuerungen auf dem Softwaremarkt zu ständig neuen Revisionen der Softwarekomponenten (*Releases*). Ohne eine strukturierte Verwaltung dieser Releases würde der Anwender schnell die Übersicht über den Zustand und die Aktualität seines Systems verlieren. Teamfähigkeit spielt insbesondere in der Einführungsphase betrieblicher Anwendungssoftwareprodukte wie dem R/3-System der SAP eine große Rolle. So müssen typischerweise Komponenten, die von mehreren Abteilungen eines Unternehmens genutzt werden – z.B. die R/3-Komponente „Materials Management“ (MM), die u.a. von der Finanzbuchhaltung, dem Einkauf und der Produktion genutzt wird – in enger Kooperation dieser Abteilungen konfiguriert werden, um unerwünschte Wechselwirkungen mit anderen Komponenten auszuschließen.

3.5 Fazit

In diesem Kapitel wurden die Grundlagen komponentenbasierter Anwendungssysteme aufbereitet. Wie herausgestellt wurde, existiert heute kein einheitlicher Komponentenbegriff. Es gibt viele verschiedene – teilweise widersprüchliche – Aspekte, die mit diesem Begriff in Verbind gebracht werden. Um im weiteren Vorgehen ein klareres Bild von dem hier zugrundegelegten Komponentenbegriff und eine Grundlage für ein Komponentenmodell zu haben, werden folgende Eigenschaften gefordert.

Komponenten

- sind binäre Softwareprodukte (Black Boxes),
- bieten direkte und (optional) indirekte Schnittstellen,
- werden in Komponenten-Frameworks eingesetzt,
- können – müssen aber nicht – beliebig auf Systemprozesse aufgeteilt werden,
- können im Vergleich zu heute vorherrschenden monolithischen Systemen leicht auf mehrere Rechner verteilt werden (Skalierbarkeit),

- basieren nicht notwendigerweise auf objektorientierter Technologie,
- werden kooperativ entwickelt und eingesetzt und benötigen deswegen Verträge („contracts“), die die Zusammenarbeit der am Entwicklungsprozess beteiligten Rollen unterstützen.

Diese Eigenschaften ergeben sich zum einen aus den Erkenntnissen der Software-Engineering-Forschung und zum anderen aus den Anforderungen betrieblicher Standardsoftwareprodukte wie sie sich z.B. heute auf dem ERP-Softwaremarkt abzeichnen.

Die Erkenntnisse im Software Engineering schlagen sich zum einen in „guten“ Entwurfsmustern (siehe [Gamma et al. 1995]) nieder, die als Grundlage für Komponenten und Komponenten-Frameworks genutzt werden, zum anderen zeichnen sich erste client- und server-seitige Komponentenstandards wie DCOM, ActiveX, JavaBeans und Enterprise JavaBeans (siehe [Tallman und Kain 1998], [SUN 1997], [Matena und Hapner 1999]), deren standardisierende Wirkung ebenfalls nicht ignoriert werden kann.

Im Bereich betrieblicher Standardsoftwareprodukte sind vor allem die sich abzeichnenden Standards wie die der Open Application Group (siehe [OAG 1999]) und anderer hersteller-übergreifender Standards wie der BMEcat (siehe [Hümpel, Renner und Schmitz 1999]) zu nennen.

Die Komponententechnologie und die dazu notwendige Standardisierung sollen dem Anwender eine höhere Flexibilität bei der Gestaltung seiner betrieblichen Anwendungssysteme bieten. Höhere Flexibilität bedeutet aber auch gleichzeitig eine größere Vielfalt an alternativen Lösungen, die systematisch strukturiert und verwaltet werden müssen. Aus diesem Grund spielt das Konfigurationsmanagement für komponentenbasierte Anwendungssysteme eine besondere Rolle. Die vorgestellten Kompositionsmodelle (Checkout/Checkin-Modell, Änderungsmengenmodell, Kompositionsmodell, Modell der langen Transaktionen) decken unterschiedliche Teilaspekte ab, die für komponentenbasierte Anwendungssysteme von Bedeutung sind. Für diese Arbeit ist die Komposition und somit das Kompositionsmodell von besonderer Bedeutung, da es die anwendungsbezogene Entwicklung und Konfiguration am besten unterstützt.

4 Sprachen zur Verhaltensbeschreibung

In Kapitel 3 wurde auf den Komponentenbegriff und ihm zugrunde liegende Erkenntnisse aus der Software-Engineering-Forschung eingegangen. In diesem Kapitel werden nun Sprachen zur Beschreibung von Komponenten untersucht. Dabei liegt ein Hauptaugenmerk auf den Verhaltenszusammenhängen von Komponenten und – im weiteren Sinne – auf Interaktionen zwischen Komponenten (und Benutzern) sowie der damit zusammenhängenden Gestaltung von Geschäftsprozessen mittels dieser Komponenten. Dabei stellt sich insbesondere die Frage, wie Interaktionsschemata (oder allgemeiner: Verhaltensschemata) grundsätzlich beschrieben werden sollen. Folgende Anforderungen soll eine Sprache zur Beschreibung dieser Zusammenhänge erfüllen. Sie soll

- kompositional sein, d.h. Verhaltensbeschreibungen sollten wie die durch sie beschriebenen Komponenten einfach und eindeutig aus anderen Verhaltensbeschreibungen zusammenzusetzen sein,
- für den Anwendungsarchitekten, den Konfigurierer und den Anwender einfach und verständlich sein und
- möglichst orthogonal sein, d.h. einen relativ geringen Sprachumfang haben.

Weitere Anforderungen, die in dieser Arbeit allerdings nur am Rande betrachtet werden, verlangen, dass

- durch deduktive Transformationen leichter über verhaltensbezogene Aussagen argumentiert werden kann und
- die maschinelle Verarbeitung von Beschreibungen und damit die Überprüfbarkeit bestimmter Eigenschaften zugesichert werden kann.

Im Abschnitt 4.1 werden zunächst Kriterien aufgeführt, nach denen Sprachen zur Verhaltensbeschreibung eingeordnet werden können. Danach werden in den Abschnitten 4.2 bis 4.4 mit einer abschließenden Bewertung in Abschnitt 4.5 drei für die Arbeit relevante Sprachklassen anhand von Beispielsprachen vorgestellt. Alle vorgestellten Sprachklassen werden heute im Umfeld komponentenbasierter Softwareentwicklung eingesetzt. Es wurden Sprachen gewählt, die jeweils Anforderungen der vorgeschlagenen Akteure (Anwender, Konfigurierer, Anwendungsarchitekten und Komponentenhersteller) widerspiegeln.

In den frühen Phasen der Softwareentwicklung (Anforderungsanalyse) werden die Geschäftsprozesse eines Unternehmens analysiert und für die Zukunft neu konzipiert. Dazu werden typischerweise natürliche oder semi-formale Sprachen wie die Ereignisgesteuerte Prozesskette (EPK) der ARIS-Methode (siehe Abschnitt 4.2) eingesetzt, die nach und nach auch an komponentenorientierte Trends (siehe z.B. [Loos und Allweyer 1998]) angepasst

wird. ARIS mit der EPK steht hier als bekanntester Vertreter einer Klasse von Sprachen, die für sich in Anspruch nehmen, von technischen Details zu abstrahieren und die Anforderungen des Anwenders in den Vordergrund zu stellen. Sie werden typischerweise von Beratern (Konfigurierern) und Anwendern eingesetzt

Die Klasse der objektorientierten Modellierungssprachen wird durch seinen zur Zeit bekanntesten Vertreter, die Unified Modeling Language (UML), beschrieben (siehe Abschnitt 4.3). Die UML hat ihren Ursprung in der objektorientierten Softwaretechnologie, wird aber ebenfalls um komponentenbasierte Aspekte erweitert. UML ist eine Sprache (oder genauer: Sprachfamilie), die typischerweise von Softwareentwicklern in den Entwurfsphasen objektorientierter Softwaresysteme eingesetzt wird. Sie betont somit die Bedürfnisse der Komponentenhersteller und teilweise auch die der Anwendungsarchitekten.

Softwarearchitekturbeschreibungssprachen sind Sprachen, die dem „Programming in the large“ dienen, also dem Programmieren auf Basis größerer, wiederverwendbarer Softwareelemente. Beim Beispiel des hier dargestellten Systems Darwin (siehe Abschnitt 4.4) handelt es sich bei den Softwareelementen um Softwarekomponenten, die als Black Boxes betrachtet werden und durch formale Kompositionsmechanismen zu größeren Systemen zusammengesetzt werden können. Darwin betont insbesondere die Bedürfnisse des Anwendungsarchitekten.

In Abschnitt 4.5 werden die vorgestellten Sprachen bezüglich der in Abschnitt 4.1 vorgestellten Kriterien bewertet und in einer Tabelle nebeneinander gestellt.

4.1 Kategorien von Verhaltensbeschreibungen

In der Informatik, der Wirtschaftsinformatik und auch der Betriebswirtschaftslehre gibt es eine Vielzahl von Sprachen zur Beschreibung des Verhaltens von Softwarekomponenten, Informationssystemen und sogar Sichten ganzer Unternehmen. Um diese Sprachen zumindest näherungsweise miteinander vergleichen sowie ihre Vor- und Nachteile bewerten zu können, bedarf es der Einführung einiger Kategorien. So werden im Folgenden unterschiedliche Paradigmen (Abschnitt 4.1.1), der Aspekt der Substituierbarkeit (Abschnitt 4.1.2), der Abstraktionsgrad einer Sprache (Unterspezifikation, Abschnitt 4.1.3), ihre Ausdrucksstärke (Abschnitt 4.1.4), ihre Verständlichkeit (Abschnitt 4.1.5), ihr Beschreibungszweck (Abschnitt 4.1.6), ihr Abstraktionsbereich (Abschnitt 4.1.7), die Beschreibungsstrukturierung (Abschnitt 4.1.8), die Kompositionalität (Abschnitt 4.1.9) und die Abstraktion durch Variablen (Abschnitt 4.1.10) betrachtet.

4.1.1 Paradigmen

Statt das Verhalten einer Komponente wie schon in Abschnitt 3.3.2 angedeutet (zumindest partiell) über Pre-/Post-Conditions zu definieren, kann man auch die Menge der gültigen

Kommunikationssequenzen zwischen einer Komponente und den sie umgebenden Komponenten definieren. Es gibt mindestens drei Arten einer solchen Definition:

- Endliche Automaten (inkl. Statecharts (siehe [Harel und Naamad 1996]) und Petri-Netze (siehe [Reisig 1986])),
- Prozessterme auf Basis von Prozesskalkülen wie CCS (siehe [Milner 1989]), CSP (siehe [Hoare 1985]) und dem π-Kalkül (siehe [Milner, Parrow und Walker 1992a]) sowie
- Trace-Spezifikationen (siehe [Olderog 1991]).

Endliche Automaten beschreiben Zustände und Geschehnisse (Transitionen). Zustände sind Zeiträume des beschriebenen Sachverhalts, in dem von den konkreten Geschehnissen abstrahiert wird. Transitionen sind Typen von Geschehnissen (Ereignisse), die von einem in einen nächsten Zustand führen.

Prozessterme abstrahieren vom Zustandsbegriff. Ein Prozessterm besteht aus einer Folge von Aktionen (Geschehnistypen), die in einer vorgegebenen Reihenfolge (auch parallel) ausgeführt werden. Allerdings wird zur Durchführung einer Aktion (z.B. Schreiben von Informationen) immer ein weiterer Prozessterm benötigt, der eine korrespondierende Aktion (z.B. Lesen von Informationen) durchführt. So ergeben beide Aktionen zusammen eine *Interaktion* (oder Kommunikation). Der Zustand eines Prozessterms (und damit der Zustand eines größeren, durch mehrere Prozessterme beschriebenen Systems) ergibt sich implizit aus der Abfolge der bisher durchgeführten Aktionen.

Da offensichtlich verschiedene Prozessterme das gleiche Verhalten aufweisen können, kann noch weiter abstrahiert werden, indem das Verhalten nur als Menge der erlaubten Traces (Spuren, Histories, siehe [Olderog 1991]) definiert wird. Ein *Trace* ist eine endliche Sequenz von Kommunikationsschritten.

In dieser Arbeit werden Prozessterme zugrunde gelegt.

4.1.2 Substituierbarkeit

Ein wesentliches Merkmal komponentenbasierter Anwendungssysteme ist die Austauschbarkeit (*Substituierbarkeit*) von Komponenten. Möchte z.B. ein Unternehmen eine Produktionsplanungskomponente, die Bestandteil des betrieblichen Anwendungssystems ist, gegen eine modernere Komponente eines anderen Herstellers austauschen, ohne zusätzlichen Programmieraufwand zu leisten, dann muss die neue Komponente die vom Anwendungssystem erwartete Struktur und Verhaltensweise haben. Diese Forderung wird als Substituierbarkeit bezeichnet. Ein Formalisierung dieses Begriffs stammt aus [Griffel 1998]: Eine Komponente K_1 ist genau dann durch eine andere Komponente K_2 *substituierbar*, wenn für jede Verwendung einer Komponente K_1 auch die Komponente K_2 eingesetzt werden kann. Es wird dann auch gesagt, dass K_2 ein Subtyp von K_1 ist.

Aus der Sicht von Traces (siehe Abschnitte 4.1.1 und 7.4) lässt sich die Substituierbarkeit von Komponenten durch andere Komponenten ebenfalls erklären: Eine Komponente ist durch eine andere genau dann substituierbar, wenn alle Traces der neuen Komponente sich auf die Traces der alten Komponente projizieren lassen. Dieses Kriterium wird auch als *Behavioural Subtyping* bezeichnet.

Analog lassen sich dann Kontrakte beschreiben, die Hersteller von Anwendungssoftwareprodukten zur Kopplung von Komponenten anderer Hersteller zur Verfügung stellen: Ein Kontrakt wird dann als abstrakte Komponente verstanden, durch die eine Menge von Traces definiert wird, die mittels einer potentiellen Komponente realisierbar sein müssen.

In [Ebert und Engels 1995] werden zwei verschiedene Arten der Beschreibung von Verhalten gefordert, nämlich von beobachtbarem Verhalten (*Observable Behaviour*) und aufrufbarem Verhalten (*Invocable Behaviour*). Die Spezifikation des beobachtbaren Verhaltens einer Komponente besagt, was eine Komponente maximal „machen“ darf. Die Spezifikation des aufrufbaren Verhaltens besagt, was eine Komponente mindestens (funktional) leisten muss. Die erste Perspektive ist aus Sicherheitsgründen interessant: Eine Komponente, die dieses Verhalten garantieren kann, „macht“ dann nicht mehr, als ihr erlaubt wurde. Die zweite Perspektive stellt sicher, dass eine geforderte Leistung einer Komponente auch wirklich erbracht wird, sie entspricht dem herkömmlichen Verständnis von Substituierbarkeit.

Die Unterschiede lassen sich an einem kleinen Beispiel (siehe UML-Diagramm in Abbildung 4.1) verdeutlichen: eine Client-Komponente benötige eine Komponente, die das Interface `IF` unterstützt. Zu diesem Interface wird u.a. eine Implementierung `IMPL` angeboten. Der Entwickler der Client-Komponente möchte nun sichergehen, dass die Komponente IMPL, die das Interface IF unterstützt, einerseits nicht irgendwelche unerlaubten Aktionsfolgen (Traces) durchführt, andererseits aber alle Aktionsfolgen durchführen kann, die unbedingt benötigt werden. Er gibt deswegen zu dem Interface IF eine Verhaltensspezifikation OS(`IF`) für das beobachtbare Verhalten (Observable Sequences) an, die alle Traces beinhaltet, die das Verhalten einer Implementierung in seinem Freiheitsgrad einschränkt und eine Verhaltensspezifikation IS(`IF`), die das mindestens erwartete Verhalten (Invocable Sequences) angibt. Soll nachgeprüft werden, ob die Komponente `IMPL` diese Anforderungen erfüllt, muss für das beobachtbare Verhalten von IMPL nachgewiesen werden[1], dass

[1] Wenn s ein Trace über die Methodensignaturen einer Menge X von Methoden und V eine Teilmenge aus X ist, dann ist s ↾ V ein Trace, der nur Methodenaufrufe enthält, die den Signaturen in V entsprechen. M(X) sind die Methodensignaturen von X. s ↾ V wird auch Projektion genannt.

$$OS(IMPL) \upharpoonright M(IF) \subseteq OS(IF)$$

gilt, und für das aufrufbare Verhalten von IMPL nachgewiesen werden, dass

$$IS(IF) \subseteq IS(IMPL)$$

gilt.

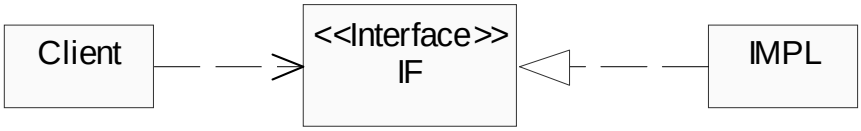


Abbildung 4.1: Substituierbarkeit

Generell können die Bedingungen überprüft werden, indem versucht wird, Homomorphismen zu finden, die die Zustände der Schnittstelle und einer Implementierung einander zuordnen. Grundsätzlich sind diese Bedingungen für turing-mächtige Sprachen zur Verhaltensbeschreibung nicht entscheidbar. Nur für Sprachen, die der Beschreibung von Systemen mit eingeschränktem Zustandsraum dienen, lassen sich diese Bedingungen automatisch überprüfen.

Im Kontext dieser Arbeit wird nur das aufrufbare Verhalten betrachtet. D.h., dass nur die Vollständigkeit bzgl. der funktionalen Anforderungen einer Implementierung interessiert, aber nicht etwaige Sicherheitsanforderungen (siehe auch Zielsetzung in Abschnitt 1.5).

Eine Verhaltensbeschreibungssprache muss folglich zur Formulierung des aufrufbaren Verhaltens gewünschter Komponenten (in Form von Kontrakten) und des tatsächlichen Verhaltens von Komponenten geeignet sein.

4.1.3 Unterspezifikation

In dieser Arbeit steht nicht die Spezifikation von Komponenten in einem formal-theoretischen Rahmen im Vordergrund (siehe auch Abschnitt 1.5). Deshalb geht es auch nicht darum, auf der Grundlage von Verhaltensbeschreibungen vollständig zu beweisen, dass eine Komponente einen vorgegeben Kontrakt erfüllt. Vielmehr soll den Nutzern einer Komponenten durch eine Beschreibung schnell ein grobes Bild der Komponente und des zukünftigen Zusammenspiels mit anderen Komponenten und Benutzern vermittelt werden.

Aus diesem Grund brauchen die Beschreibungen von Komponenten und Kontrakten im Verhältnis zu den tatsächlichen Realisierungen – den Komponenten – nur Unterspezifikationen zu sein. Die Unterspezifikation kann insbesondere durch unvollständige Beschreibung von Methoden- und Prozedurparameterwerten sowie durch nicht-deterministische Entscheidungen an Auswahlpunkten (choices) zustande kommen.

Der Hauptvorteil unterspezifizierter Verhaltensbeschreibungen liegt in der Abstraktion von unwichtigen Details und damit gleichzeitig in der leichteren Verständlichkeit.

Ein Nachteil ist offensichtlich die Tatsache, dass nicht ausgeschlossen werden kann, dass eine Komponente, deren Beschreibung die Beschreibung eines Kontraktes substituiert, in Wirklichkeit den Kontrakt doch nicht einhält. Um dennoch einen 100%-igen Substituierbarkeitstest vollziehen zu können, müssen andere Dokumente der Softwareentwicklung als die hier behandelten Komponentenbeschreibungen herangezogen werden. Hierbei gerät man jedoch grundsätzlich schnell an die Grenzen dessen, was automatisch zu beweisen ist (siehe nächsten Abschnitt).

Wieweit unterspezifiziert, also von einer vollständigen formalen Beschreibung abstrahiert wird, muss letztlich vom Zweck der Beschreibung abhängig gemacht werden. Ein Beispiel sind die Beschreibungen der Ereignisgesteuerten Prozessketten des R/3-Referenzmodells (siehe Abschnitt 4.2). Diese sind ebenfalls unterspezifiziert, da sie das Verhalten des R/3-Systems nur bruchstückhaft wiedergeben. Dennoch erfüllen Sie den Zweck, den Anwendern und Beratern einen ausreichenden Überblick über die Funktionalität und die unterstützten Geschäftsprozesse des R/3-Systems zu geben.

4.1.4 Ausdrucksstärke (Expressive Power)

Eine (künstliche) Sprache ist *ausdrucksstark*, wenn sie es erlaubt, alle geforderten Sachverhalte möglichst einfach darzustellen. Oft fordert man neben der Ausdrucksstärke auch noch, dass der Sprachumfang möglichst klein ist (*Orthogonalitätsprinzip*), es aber dennoch möglich ist, alle geforderten Sachverhalte *effektiv* auszudrücken.

Die Ausdrucksstärke einer Sprache zur Beschreibung von Komponenten kann aus mindestens zwei Perspektiven betrachtet werden – der formalen und der inhaltlichen Perspektive. *Formale Ausdrucksstärke* kann im Umfeld der Informatik vor allem am Berechenbarkeitsbegriff festgemacht werden. Vor dem Hintergrund der Churchschen These (siehe z.B. [Engesser, Claus und Schwill 1993]) hat eine Programmiersprache dann die größte Ausdrucksstärke, wenn mit ihr alle berechenbare Funktionen formuliert werden können, wobei der Berechenbarkeitsbegriff durch die Turing-Mächtigkeit formalisiert wird. Ein effektiver Beweis für die Turing-Mächtigkeit kann für eine Sprache dadurch erbracht werden, dass mittels der Programmiersprache eine Turingmaschine implementiert wird. Ähnliche Einordnungen der Ausdrucksstärke sind auch entlang der Chomsky-Hierarchie (siehe [Engesser, Claus und Schwill 1993]) für Grammatiken möglich.

In formalen Sprachen werden Elementaraussagen wie in der Logik meist nur als bloße Zustandsaussagen in Tupelschreibweise mittels formaler, inhaltsloser Symbole und darauf basierender Transformationen formuliert. Um eine Zuordnung zur „realen“ Welt herzustellen, bedarf es einer *Interpretation* dieser Tupel und Symbole. In formalen Systemen werden zu

den Symbolen und Tupeln typischerweise extensionale Interpretationen in Form von Mengen angegeben. Eine solche Interpretation ist aus formaler Sicht für ein Anwendungssystem heute kaum zu leisten, da der Aufwand in keinem wirtschaftlichen Verhältnis zum Nutzen steht.

Während in der Theoretischen Informatik z.B. durch explizite Angabe einer Interpretation geprüft werden kann, ob eine Aussage wahr oder falsch ist, wird im Bereich der Wirtschaftsinformatik auf eine eher intuitive Semantik gesetzt. Es findet keine explizite Interpretation zwischen Formel und formuliertem Weltausschnitt statt. Diese Zuordnung wird statt dessen implizit vorgenommen, indem anstelle der Symbole Prädikatoren benutzt werden, für die beim Leser eine allgemeingültige Bedeutung vorausgesetzt wird. Diese Form der Interpretation ist problematisch, da nicht immer davon ausgegangen werden kann, dass ein Prädikator bei jedem Leser auch die gleiche Bedeutung hat.

Eine Sprache zur Beschreibung des Verhaltens von Komponenten sollte also einerseits eine ausreichend formale Ausdrucksstärke besitzen und andererseits den Interpretationsprozess für den Leser möglichst einfach gestalten. Dies ist in der Tat eins der Hauptprobleme der Informatik, das in vielen Anwendungsbereichen immer wieder auftritt.

4.1.5 Verständlichkeit

Die Verständlichkeit einer Sprache ist eine Eigenschaft, die man nur empirisch nachweisen kann. Eine Untersuchung dieser Eigenschaft ist nicht Bestandteil dieser Arbeit. Jedoch können Vermutungen angeführt werden, wann eine Sprache verständlicher und schneller zu verstehen ist als eine andere.

Wichtig ist im Umfeld künstlicher Sprachen, dass die Sprachelemente (Symbole, Satzformen) schnell erlernt und interpretiert werden können. Dieser Arbeit liegt die Idee (siehe Hypothese 5 in Abschnitt 1.4) zugrunde, dass diese Interpretation wesentlich vereinfacht wird, wenn man bei der Definition der Sprache möglichst direkt auf die natürliche Sprache zurückgreift. Wie schon in Abschnitt 1.3 dargestellt, stellt sich bei der Definition jeder Sprache die Frage nach dem methodischen Ursprung: Womit bzw. auf welcher Grundlage wird eine Sprache definiert? Welche (Sprach-)Mittel können für die späteren Anwender dieser Sprache vorausgesetzt werden? Es wird hier der radikalen Sprachkritik gefolgt, wie sie von Lorenzen u.a. (siehe z.B. [Lorenzen 1987]) gefordert wird. Dem sprachkritischen Programm liegt die Erkenntnis (siehe Abschnitt 1.3 und [Quine 1960], [Wedekind 1992]) zugrunde, dass eine Sprache nicht ohne ein vorhandenes Fundament – nämlich letztlich der schon bestehenden, natürlichen Sprache – aufgebaut werden kann und deshalb versucht werden muss, dieses Fundament systematisch zu nutzen und zu festigen. Das Festigen kann nur so vonstatten gehen, dass die Sprache auf die Grundbegriffe zurückgeführt wird, über deren Bedeutung in einer Sprachgemeinschaft kein Zweifel mehr besteht. Darauf aufbauend kann nach und nach eine (Fach-)Sprache definiert werden. Die Orthosprache von Lorenzen ist eine solche

Sprache, und sie hat zudem den Vorteil, dass ihre Syntax sich an den germanischen Sprachen (z.B. Englisch und Deutsch) orientiert.

In dieser Arbeit wird von der These ausgegangen, dass eine Sprache, die methodisch rekonstruiert wurde, leichter zu verstehen ist, als eine (formale) Sprache, die einfach „gegeben“ (siehe [Hartmann 1990]) wird. Sie ist ebenfalls leichter zu verstehen als die sogenannten semi-formalen Sprachen wie die EPK, die zwar inhaltlich durch ihr Anwendungsgebiet geprägt sind, aber ständig erweitert werden müssen, weil sie eben nicht systematisch und konstruktiv aufgebaut wurden.

4.1.6 Beschreibungszweck: ergebnis- versus prozessorientiert

Verhaltensbeschreibungen können ergebnisorientiert oder prozessorientiert (siehe [Wendt 1998]) sein. *Ergebnisorientierung* liegt vor, wenn es demjenigen, der einen Nutzen von dem System haben will, genügt, eine Anfangssituation und eine Endsituation zu betrachten. Ergebnisorientierte Sprachen dienen der Formulierung von Funktionen. Herstellende (poietische) Handlungen wie die Produktion in einem Unternehmen sind typischerweise ergebnisorientiert.

Prozessorientierung liegt vor, wenn der Nutzen des Systems im Prozess selbst liegt. Dabei ist es völlig unerheblich, ob in diesem Prozess ein Ergebnis geschaffen wird oder nicht. Dienstleistungen sind typischerweise prozessorientiert.

Es besteht oft das Interesse, Systeme sowohl aus ergebnisorientierter als auch aus prozessorientierter Sicht zu betrachten. Im betrieblichen Umfeld wurde traditionell ergebnisorientiert gedacht, z.B. an die Gewinnmaximierung als Unternehmensziel. Die prozessorientierte Betrachtung (und damit Beschreibung) hielt erst Einzug in das Unternehmen (siehe [Hammer und Champy 1994]), als erkannt wurde, dass die Leistungen eines Unternehmens nicht mehr nur nach deren Output beurteilt werden darf, sondern auch Faktoren wie prozessbegleitendes Qualitätsmanagement und Kundenorientierung einen starken Einfluss auf den Unternehmenserfolg haben.

Zur Beschreibung des Verhaltens einer Komponente müssen sowohl Ergebnis- als auch die Prozessorientierung berücksichtigt werden. Ergebnisorientierung spielt insbesondere bei der isolierten Beschreibung der Komponenten und ihrer Operationen eine Rolle. Die Prozessorientierung kommt dann ins Spiel, wenn man das Zusammenspiel der Komponenten im Rahmen der Geschäftsprozesse eines Unternehmens betrachtet.

4.1.7 Abstraktionsbereich: geschehnis- versus zustandsorientiert

Sprachen zur Beschreibung von Geschehnissen (vgl. hierzu auch Abschnitt 4.1.1) werden in der Regel aus höchstens zwei Grundelementen aufgebaut. Das eine dient der Beschreibung

der eigentlichen Geschehnisse (Taten, Handlungen, Interaktionen usw.), das andere dient der Beschreibung von Zuständen. Zustände sind (siehe [Lorenzen 1987]) Augenblicke zwischen verschiedenen Geschehnissen, in denen vom Geschehen abstrahiert wird – in denen also das Geschehen zum Zwecke der Beschreibung nicht interessiert. Sie sind also relativ „stabile Geschehnisse“. Zustände können zu komplexeren Zuständen zusammengefasst werden, die dann auch als Situationen bezeichnet werden. In den Sprachen zur Verhaltensbeschreibung bekommen die beiden Sprachelemente eine unterschiedliche Gewichtung (vgl. auch 5.10). So abstrahieren Sprachen für die Modellierung von Geschäftsprozessen wie z.B. die Activity Charts der UML von den Zuständen und stellen nur die Geschehnisse in den Mittelpunkt der Betrachtung. Andere Sprachen wie Statecharts stellen sowohl Zustände wie auch Transitionen (= bedingte Geschehnisse) zur Beschreibung der Übergänge zur Verfügung. Jedoch liegt auch hier das Hauptgewicht auf der Beschreibung der Transitionen.

Sogenannte *buchstabierende Automaten*, wie sie in [Rumpe 1997] genutzt werden, stellen das andere Extrem dar. Hier beschränken sich die Beschreibungen der Transitionen (= Geschehnisse) auf einen Buchstaben, der eine Aktion symbolisiert. Dafür werden jedoch die Zustände ausführlicher beschreiben. Durch diese deklarative Art der Beschreibung – es wird beschrieben, was vor und nach einer Transition gelten muss, aber nicht wie die Transition funktionieren muss – sollen in den frühen Entwicklungsphasen Implementierungsdetails, die bei imperativen Programmiersprachen ja eher den Aktionen entsprechen, nicht vorweggenommen werden.

Die Ereignisgesteuerte Prozesskette stellt – ähnlich wie auch die Petri-Netze (siehe [Reisig 1986]) – für beide Aspekte Sprachelemente zur Verfügung. *Funktionen* dienen der Beschreibung von Geschehnissen und *Ereignisse* (im Sinne der EPK, siehe 4.2.2) dienen der Beschreibung von Zuständen. Hier wird offensichtlich, dass das Wort „Ereignis“ für einen Zustand im Kontext der EPK unglücklich gewählt wurde.

Zusammenfassend ist zu bemerken, dass man sich bei der Konstruktion einer Sprache zur Beschreibung des Verhaltens von Komponenten grundsätzlich der beiden Kategorien Geschehnis und Zustand bewusst sein muss, um zu entscheiden, was als Geschehnis und was als Zustand abgebildet und auf welches der beiden Konzepte das Hauptaugenmerk gelegt werden soll.

4.1.8 Strukturierung: objekt- versus vorgangsorientiert

Möchte man mehrere inhaltlich zusammenhängende Geschehnisschemata eines Sachverhaltes zusammenfassen, so stellt sich die Frage, wie die Beschreibungen strukturiert werden. In der Informatik gibt es mindestens den objekt- und den vorgangsorientierten (auch: imperativer) Ansatz der Strukturierung:

1. Im objektorientierten Ansatz werden alle Beschreibungen von Geschehnisschemata, an denen ein oder mehrere Objekte beteiligt sind, in der Hauptsache genau einem der zugrunde liegenden Objektschemata (= Klasse) zugeordnet. Welche Geschehnisschemata einem Objektschema zugeteilt werden, hängt von der Zielsetzung ab, die durch die Entwicklung eines System verfolgt wird. Beispielsweise würde ein Geschehnisschema für „Unfälle“ in einer Softwarekomponente, die der Unterstützung eines Abschleppdienstes dient, wahrscheinlich eher einem Objektschema für Autos zugeordnet werden, während es in einer Komponente für Krankentransportunternehmen einem Schema für „Unfallopfer“ zugeteilt würde. In der objektorientierten Programmierung werden diese Geschehnisschemata als *Prozeduren* (oder *Methoden*) bezeichnet. In der Entwurfsphase objektorientierter Systeme werden die Geschehnisschemata und ihre Abhängigkeiten voneinander durch Zustandsmaschinen, wie sie z.B. Harels Statecharts (siehe [Harel und Naamad 1996]) zugrunde liegen, beschrieben. In [Schienmann 1997] werden solche Arten von Beschreibungen auch als *Wandlungen* bezeichnet.

2. Im vorgangsorientierten Ansatz werden die Beschreibungen von Geschehnissen Vorgängen (genauer: Vorgangsschemata) zugeordnet. Die Vorgänge – im betrieblichen Umfeld meist als *Geschäftsprozesse* bezeichnet – richten sich an vorgegebenen (Geschäfts-)Zielen aus. An einem Vorgang sind mehrere Objekte (Akteure) beteiligt, die kooperieren, d.h. zielorientiert miteinander handeln (interagieren). Ein typischer Geschäftsprozess ist z.B. die Kundenauftragsabwicklung in einem Unternehmen, die die Schritte (Geschehnis- bzw. Handlungsschemata) von der Auftragsannahme bis zu Auslieferung an den Kunden beinhaltet. Zu den Beschreibungssprachen, die den vorgangsorientierten Ansatz verfolgen, gehören beispielsweise die Ereignisgesteuerte Prozesskette (siehe Abschnitt 4.2) und die Activitycharts der UML (siehe [Booch, Rumbaugh und Jacobson 1997a]). In [Schienmann 1997] werden solche Arten von Beschreibungen auch als *Reihenfolgen* bezeichnet.

Was die formale Ausdrucksstärke der beiden Ansätze (siehe Abschnitt 4.1.4) angeht, sind keine prinzipiellen Unterschiede festzustellen. Es gibt sowohl objektorientierte (z.B. Statecharts) als auch vorgangsorientierte Sprachen (z.B. Petri-Netze), die turing-mächtig sind. Was die Effektivität und Verständlichkeit der Sprachen angeht, ist zu vermuten, dass objektorientierte Sprachen den vorgangsorientierten Sprachen überlegen sind. So besteht zur Zeit ein Trend[1] zu objektorientierten Sprachen. Allerdings bieten moderne Methoden des Software Engineerings wie UML für beide Kategorien Sprachen an.

[1] Ein Beispiel für diesen Trend sind die Bemühungen, die Ereignisgesteuerte Prozesskette in die objektorientierte Welt zu überführen (siehe [Loos und Allweyer 1998]).

4.1.9 Kompositionalität

Ein wesentliches Kriterium für eine Verhaltensbeschreibungssprache ist die Kompositionalität. *Kompositionalität* heißt, dass die Bedeutung eines komplexen Ausdrucks sich aus der Bedeutung der Einzelausdrücke und der Struktur des Gesamtausdrucks eindeutig ableiten lässt. So ist die Syntax der deutschen Sprache grundsätzlich nicht kompositional. Es existieren *strukturelle Ambiguitäten*, die das eindeutige Verstehen eines Satzes zu einem Problem machen. So hat beispielsweise der Satz „Der Mörder erwürgte den Mann mit der roten Krawatte“ offensichtlich zwei unterschiedliche Bedeutungen. Die Bedeutung dieses Satzes lässt sich also nicht aus den einzelnen Teilen und der verwendeten Struktur ableiten.

Trivialerweise bedingt die Forderung nach Kompositionalität, dass es überhaupt Strukturelemente (Operatoren) gibt, mit deren Hilfe komplexe Ausdrücke aufgebaut werden können, und dass es klare Regeln gibt, nach denen diese komplexen Ausdrücke interpretiert werden können. In einigen Arbeiten – beispielsweise rund um die Ereignisgesteuerte Prozesskette (EPK) – werden zusätzlich neue, nicht zur EPK gehörende Sprachkonstrukte wie z.B. die *Prozesspartikel* in [Remme und Scheer 1996] entwickelt. Diese Ansätze erfordern neue, zusätzliche Kompositionssprachen, deren formales Fundament zusätzlich erarbeitet werden muss – aber oft nicht erarbeitet wird.

Die Kompositionalität ist insbesondere im Hinblick auf eine Sprache zur Beschreibung des Verhaltens von Softwarekomponenten von besonderer Bedeutung, da man auch von den beschriebenen Objekten (Softwarekomponenten) Kompositionalität fordert. So sollte eine Beschreibung einer komplexen Komponente einfach aus den Beschreibungen der Subkomponenten zusammengesetzt und die Bedeutung einer solchen komplexen Komponente möglichst einfach analysiert werden können.

Formaler betrachtet bedeutet das, dass eine Komposition zweier Komponentenbeschreibung wieder eine gültige Komponentenbeschreibung ist, und dass diese Beschreibung natürlich auch die resultierende komplexe Komponente semantisch korrekt beschreibt.

4.1.10 Variablen

Je nach Zweck und Ausrichtung einer Sprache zur Verhaltensbeschreibung wird von möglichen konkreten Begebenheiten eines zu beschreibenden Sachverhalts abstrahiert. So abstrahieren Sprachen für die Beschreibung von Geschäftsprozessen wie die Ereignisgesteuerte Prozesskette (EPK) vollkommen von den im Rahmen eines Sachverhalts auftretenden Individuen, da es keine Variablen zur Referenzierung von Individuen gibt. Alle konkreteren Aussagen über Sachverhalte können nur durch zusätzliche, mehr oder weniger formalisierte Beschreibungen innerhalb der EPK-Sprachkonstrukte (natürlichsprachliche Bezeichnungen und Erklärungen von Funktionen und Ereignissen) getroffen werden.

Andere Sprachen wie beispielsweise die Statecharts beschreiben referenzierbare Aussagen mittels Variablen und sind somit wesentlich konkreter.

4.2 Sprachen zur Geschäftsprozessmodellierung

Es gibt eine Vielzahl von Ansätzen zur Beschreibung von Geschäftsprozessen (vgl. z.B. [Ferstl und Sinz 1998], [Jaeschke 1995], [Langer 1994] und [Scheer 1998a]). Die bekannteste Sprache ist sicherlich die *Ereignisgesteuerte Prozesskette* (EPK) aus der *ARIS*-Methode (siehe [Scheer 1998a] und [Scheer 1998b]). ARIS und die EPK werden in den nächsten Abschnitten als typische Vertreter dieser Klasse von Sprachen vorgestellt. Die EPK ist u.a. als Beschreibungssprache für das R/3-Referenzmodell der SAP (siehe z.B. [Keller und Teufel 1997]) bekannt geworden.

4.2.1 ARIS

ARIS basiert auf der Idee, dass die an Geschäftsprozessen beteiligten Objekte (Akteure, Anwendungssysteme, Daten, Informationen) isoliert voneinander beschrieben werden können, um dadurch die Komplexität der Darstellung der Zusammenhänge aller Objekte beherrschbar zu machen. In ARIS werden die Geschäftsprozesse und die daran beteiligten Objekte deshalb in vier Sichten (siehe Abbildung 4.2) aufgeteilt, und zwar in

- die *Organisationssicht*, in der die Mitarbeiter (menschliche Akteure) und ihre Organisationsstrukturen beschrieben werden,
- die *Datensicht*, in der die Daten bzw. Informationen und ihre Abhängigkeiten untereinander beschrieben werden,
- die *Funktionssicht* (oder Aufgabensicht), in der die Aufgaben in ihrer hierarchischen Abhängigkeitsstruktur – eine Aufgabe kann aus mehreren Teilaufgaben bestehen – dargestellt werden, und
- die *Prozesssicht* (oder Steuerungssicht), in der die Geschäftsprozesse unter Rückgriff auf die in den anderen Sichten definierten Objekte integriert dargestellt werden.

In neueren ARIS-Versionen (siehe [Scheer 1998a]) kommt noch eine fünfte Sicht[1], die Leistungssicht, hinzu. Sie dient der Beschreibung der betrieblichen Leistungen (Produkte, Sachleistungen, Dienstleistungen) beispielsweise mittels Produktdatenmodellen.

---

[1] Jede Einteilung in Sichten ist abhängig von der Zielsetzung, die man mit den erstellten Beschreibungen verfolgt. So sind weitere Sichten – etwa eine Ressourcensicht für Maschinen und Anlagen – denkbar, umgekehrt müssen auch nicht alle vorhandenen Sichten für eine angestrebte Beschreibung genutzt werden. Gemessen an dieser Flexibilität und Dynamik ist das Sichtenkonzept von ARIS mit fünf Sichten relativ statisch.

Da in dieser Arbeit besonders der Verhaltensaspekt betont wird, wird zunächst die Prozesssicht von ARIS betrachtet. Prozesse werden in ARIS auf Grundlage der Ereignisgesteuerten Prozesskette (EPK) beschrieben.

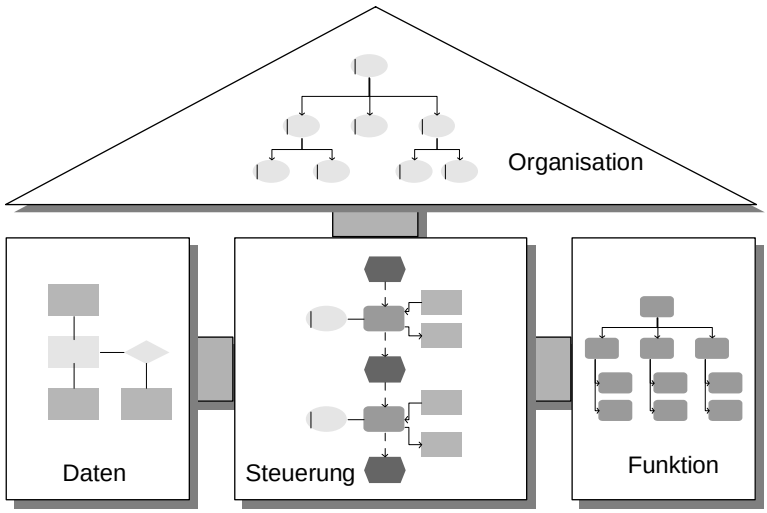


Abbildung 4.2: Skizze der ARIS-Architektur

Jede EPK wird als Diagramm abgebildet, die angestrebte vorherrschende Leserichtung ist dabei von oben nach unten. Die wesentlichen Diagrammelemente (Knoten- und Kantentypen) für EPKn werden in der Abbildung 4.3 aufgelistet. *Funktionen* bezeichnen fachliche Aufgaben oder Tätigkeiten, die im Rahmen der (Inter-)Aktionen von einem oder mehreren Objekten durchgeführt werden. Sie werden graphisch als grüne Rechtecke mit abgerundeten Ecken dargestellt. Funktionen erzeugen Ereignisse, auf die wiederum in anderen Funktionen reagiert werden kann. Der Begriff „Ereignis“ ist nicht immer ganz treffend, in vielen Fällen sollte man besser von *Zuständen* sprechen, die das Ausführen weiterer Funktionen ermöglichen bzw. voraussetzen. Eine Funktion wird immer von einem oder mehreren Ereignissen (Startereignissen) ausgelöst und erzeugt als Ergebnis ein oder mehrere Ereignisse (Endereignisse). *Ereignisse* spezifizieren also betriebswirtschaftlich relevante Zustände von Informationsobjekten, die vor der Durchführung von Funktionen erfüllt sein müssen. Andererseits werden durch die Durchführung dieser Funktionen die Zustände von Objekten verändert. Dadurch können andere „anschließende“ Funktionen durchgeführt werden usw. Ereignisse werden graphisch durch rote Sechsecke dargestellt. Da Ereignisse Funktionen auslösen und außerdem Ergebnisse von Funktionen sind, entsteht eine alternierende Folge von Ereignissen und Funktionen, die mit mindestens einem Ereignis startet und mit mindestens einem Ereignis endet. Die Folge aus Ereignissen und Funktionen wird mit gestrichelten, gerichteten Kanten zu einem Prozess verbunden, woraus die zeitlich-logische Abfolge der Funktionen hervorgeht. Die Menge der Kanten beschreibt damit den Kontrollfluss in einem Prozess.

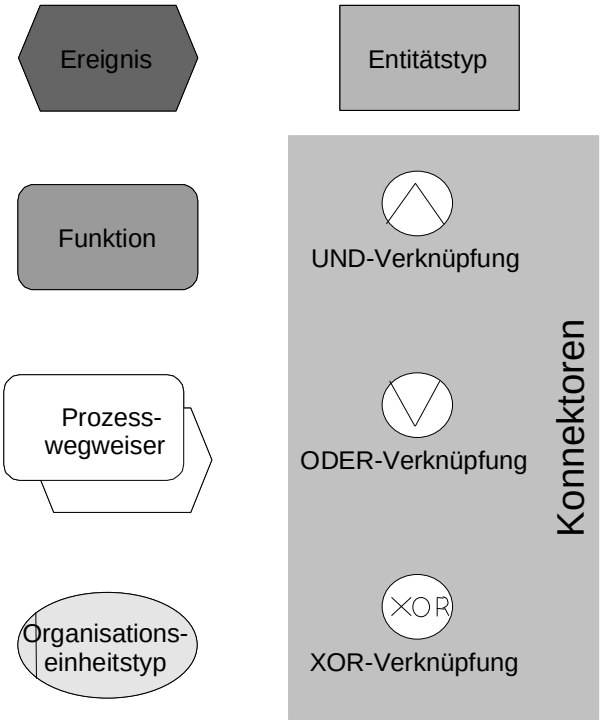


Abbildung 4.3: Beschreibungselemente des EPK-Modells

Um möglichst viele Varianten ähnlicher Geschäftsprozesse in einem Diagramm darstellen zu können, enthält das EPK-Modell neben den Funktionen und Ereignissen noch *Verknüpfungsoperatoren*, auch *Konnektoren* genannt, die es ermöglichen, alternative (OR-Operator), exklusiv-alternative („Entweder-Oder“, XOR-Operator) und parallele Abfolgen (AND-Operator) von Aktionen zu beschreiben. Die Verknüpfungsoperatoren werden als weiße Kreise dargestellt, die das Symbol des logischen Operators beinhalten. Die Verknüpfungen des Steuerflusses zwischen den Ereignissen und Funktionen können in Ereignis- und Funktionsverknüpfungen differenziert werden. Bei Ereignisverknüpfungen wird der Steuerfluss zweier oder mehrerer Ereignisse mit einer vor- oder nachgelagerten Funktion mit Hilfe eines Verknüpfungsoperators aufgesplittet bzw. zusammengefasst. Bei den Funktionsverknüpfungen wird als Resultat von zwei oder mehr Funktionen ein Ereignis modelliert und der Steuerfluss mit dem entsprechenden Operator verknüpft. Außerdem kann ein Ereignis mit zwei oder mehr nachfolgenden Funktionen mit Hilfe einer konjunktiven Verknüpfung verbunden werden. Andere Verknüpfungsoperatoren dürfen nicht verwendet werden, da Ereignisse keine „Entscheidungskompetenz“ besitzen, d.h. ein auslösendes Ereignis nur für alle ihm folgenden Funktionen auslösend sein kann. Komplexere logische Strukturen können durch die Verknüpfung des Steuerflusses mit Hilfe mehrerer, aufeinander folgender Operatoren modelliert werden.

Die Funktionen eines Geschäftsprozesses können auf verschiedenen Abstraktionsebenen betrachtet werden. So ist es möglich, mehrere Aktionen einer EPK hinter einer Funktion zu

verbergen, um z.B. nebensächliche Details zu „verstecken“. Diese Art der Abstraktion von Details kann mehrfach wiederholt werden, so dass hinter Funktionen einer EPK wieder EPKn liegen, die ebenfalls Funktionen besitzen, hinter denen wieder EPKn liegen usw.

Zur Darstellung von Abhängigkeiten zwischen EPKn können *Prozesswegweiser* eingefügt werden. Diese Wegweiser verweisen auf EPKn, die entweder vor oder nach der aktuellen EPK ausgeführt werden. Sie spiegeln also die Verbindungsinformationen einer anderen, typischerweise nächsthöheren Abstraktionsebene wider.

Zur Darstellung der an einer Interaktion beteiligten Objekte (Akteure) werden Knoten vom Typ *Organisationseinheit* angeboten, die die Abteilung, die Gruppe oder die Stelle eines beteiligten Akteurs repräsentieren, wodurch sinnvollerweise von den konkreten Mitarbeitern des Unternehmens abstrahiert wird. Zur Darstellung der verbrauchten, erzeugten und transformierten Objekte werden sogenannte *Entitätstypen* (= Typen von „Dingen“) angeboten. Diese Entitätstypen werden gleichzeitig in der Datensicht verwaltet.

Um grundsätzlich über Organisationseinheiten sprechen zu können, macht es Sinn, von *Organisationseinheitstypen* zu sprechen, zu denen z.B. Abteilungen, Gruppen, Projektteams und Stellen gehören. Genau genommen sind Organisationseinheitstypen nichts anderes als Entitätstypen, da Ausprägungen (konkrete Entitäten, wie die Finanzabteilung, die Einkaufsorganisation usw.) davon ebenfalls „Dinge“ sind, die in dem Geschäftsprozess der Unternehmensgestaltung erzeugt, verändert und gelöscht werden. Obwohl die Organisationseinheitstypen als Ausschnitt der Datensicht zu verstehen sind, werden sie in ARIS zusätzlich in der Organisationssicht verwaltet.

4.2.2 Ereignisgesteuerte Prozesskette

Der Begriff „Geschäftsprozess“ wurde schon benutzt, aber noch nicht präzise beschrieben. Allgemein versteht man unter einem *Geschäftsprozess* eine Abfolge von Aktionen bzw. Interaktionen, die von Objekten bzw. zwischen mehreren Objekten durchgeführt werden und einem betrieblichen Ziel dienen. Die Akteure, also die Objekte, die die Aktionen durchführen, sind meist Mitarbeiter, Kunden, Lieferanten und auch technische Systeme wie Anwendungssysteme oder Produktionsanlagen des Unternehmens. Bei der Durchführung der Aktionen werden Objekte verbraucht (Input), erzeugt (Output) und/oder transformiert (In- und Output). Diese Objekte können Materialien, Produkte, Informationen und allgemein Dienstleistungen sein.

In Abbildung 4.4 ist eine kleine Beispiel-EPK (sie wurde teilweise dem R/3-Referenzmodell entnommen) zu sehen, die typische Aufgabenzusammenhänge des Beschaffungswesens beschreibt. Nach Auftreten von (Material-)Bedarfen in den Fachabteilungen eines Unternehmens werden entweder unternehmensinterne Bestellanforderungen manuell eingegeben oder in angebundenen Systemen (hier Fertigung und Instandhaltung) automatisch

generiert. Zur Beschaffung stehen der Einkaufsabteilung zwei Alternativen (XOR) zur Verfügung. Das gewünschte Material wird entweder über einen bestehenden Lieferplan oder durch Erstellung einer einfachen Bestellung beschafft. Im zweiten Fall muss die Bestellung vor ihrer Übermittlung genehmigt werden. Trifft danach die gewünschte Ware ein, so wird sie durch die Warenannahme entgegengenommen und kontrolliert. Der Wareneingang wird dazu gebucht und an die Fertigung/Instandhaltung und die Rechnungsbearbeitung weitergeleitet. Die Finanzabteilung weist die inzwischen eingetroffene Rechnung zur Zahlung durch einen automatischen Zahlungslauf an.

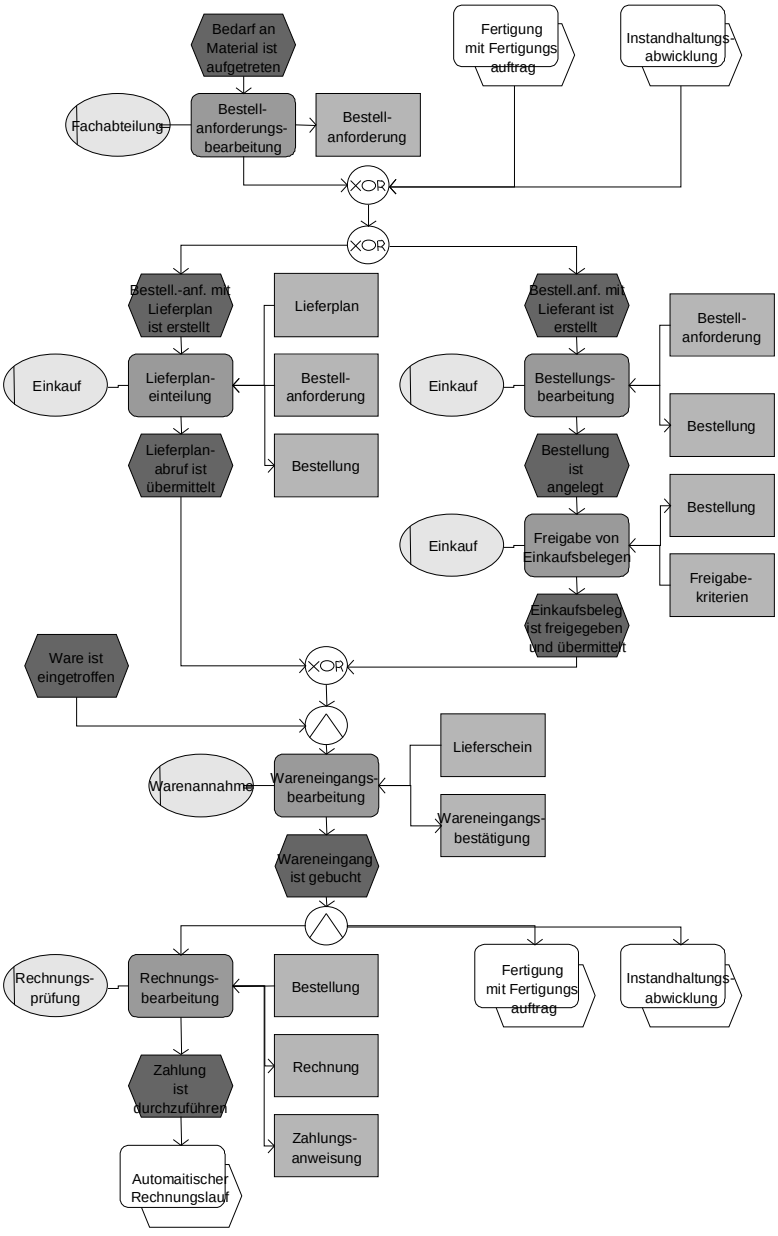


Abbildung 4.4: EPK-Beispiel

4.2.3 Softwarereferenzmodelle

Softwarereferenzmodelle sind formale oder zumindest semi-formale Beschreibungen von betrieblichen Anwendungssoftwareprodukten, die dem Anwendungsberater und dem Anwender bei der Einführung und der Konfiguration dieser Softwareprodukte helfen sollen.

Allgemein lassen sich Referenzmodelle (siehe [Winter et al. 1998]) als Modelle (= Beschreibungen) für eine Menge S von Systemen verstehen, die sich

- als Grundlage für die Konstruktion spezieller Modelle von Systemen aus S und/oder
- als Vergleichsobjekt für Modelle von Systemen aus S

nutzen lassen.

Unter *Modellen* werden in dieser Arbeit Beschreibungen von Systemen verstanden, die mit formalen oder semi-formalen Diagrammsprachen verfasst wurden. Beispiele für solche Sprachen sind die EPK der ARIS-Methode und die UML (siehe Abschnitt 4.3).

Im Umfeld der betrieblichen Unternehmensorganisation gibt es neben den Softwarereferenzmodellen eine Vielzahl von weiteren Typen von Referenzmodellen. In [Winter et al. 1998] werden *Organisations-*, *Informations-*, *Software-* und *Vorgehensreferenzmodelle* unterschieden.

- Organisationsreferenzmodelle stellen die umfassendste Art von Referenzmodellen dar. Mittels dieser Referenzmodelle wird die Begriffswelt einer Organisation (z.B. eines Unternehmens) definiert. Dazu werden typische Aufbau- und Ablauforganisationen beschrieben, wie sie beispielsweise in einer Branche (Branchenreferenzmodell) vorkommen.
- Informationssystemreferenzmodelle betrachten das betriebliche Informationssystem, also das „Nervensystem“ eines Unternehmens, und abstrahieren damit von dem Basissystem (siehe Abschnitt 2.1.2) des Unternehmens. Es werden also vornehmlich Strukturen und Geschäftsprozesse betrachtet, die der Informationsverarbeitung dienen.
- Softwarereferenzmodelle beschreiben Softwareprodukte und damit letztlich die Menge der aus ihnen konkret ableitbaren Konfigurationen in Form betrieblicher Anwendungssysteme. Bekanntestes Beispiel ist das R/3-Referenzmodell (siehe [Keller und Teufel 1997]), das die zur Verfügung stehenden Datenstrukturen in Form von Entity-Relationship-Modellen, die R/3-Transaktionen mit ihren einzelnen Benutzeraktionen als EPKn und die durch die Software unterstützten Geschäftsprozesse ebenfalls als EPKn beschreibt.
- Vorgehensmodelle stehen orthogonal zu den bisher dargestellten Referenzmodellen. Während durch die bisherigen Referenzmodelle der Zustand eines Systems (Unternehmen, betriebliches Informationssystem, Anwendungssystem) für eine Situation (Ist- oder Soll)

beschrieben wird, beschreiben Vorgehensreferenzmodelle typische Projektstrukturen und -abläufe, wie sie für konkrete Projekte – beispielsweise die Einführung eines Anwendungssoftwareprodukts – benötigt werden.

4.3 Sprachen zur objektorientierten Softwareentwicklung

In diesem Abschnitt wird die *Unified Modeling Language* (UML, siehe [Booch, Rumbaugh und Jacobson 1997a], [Booch, Rumbaugh und Jacobson 1997b]) als der im Augenblick bekannteste Vertreter einer Modellierungssprache für die objektorientierte Softwareentwicklung vorgestellt. UML ist das Ergebnis einer Verschmelzung einer Reihe verschiedener Sprachansätze (siehe [Booch 1994], [Jacobson 1998], [Rumbaugh 1991]).

UML ist eine Kombination von Teilsprachen, die der Entwicklung objektorientierter Softwaresysteme dienen. Einige dieser Teilsprachen, wie z.B. Use-Case- und Sequenzdiagramme, werden in den frühen Phasen der Softwareentwicklung (Ist- und Anforderungsanalyse) eingesetzt. Andere Teilsprachen, beispielsweise die Klassen- und Zustandsdiagramme, werden auch in der Analyse-, aber vor allem in der Entwurfsphase verwendet (siehe dazu [Fowler und Scott 1998]). Bedingt durch die Ansprüche der verschiedenen Softwareentwicklungsphasen werden die UML-Teilsprachen, besonders die Klassendiagramme, in unterschiedlichen Abstraktionsgraden eingesetzt. So können die konzeptuelle Sichtweise (oft auch semantische oder fachliche Sicht genannt) der Analysephase und die eher logische Entwurfssicht unterschieden werden. Während bei der ersten Sichtweise die Modellierung des Diskursbereiches im Vordergrund steht, werden bei der zweiten Sichtweise die Struktur und das Verhalten des zukünftigen Softwaresystems anvisiert. Um ein objektorientiertes Softwaresystem vollständig zu beschreiben, reicht eine Teilmenge der Sprachen vollkommen aus. Ausgeschlossen werden können die Sprachen, die ausschließlich der Analysephase dienen, und die sogenannten „reflective languages“, die der zusätzlichen Dokumentation des Softwaresystems aus weiteren Blickwinkeln dienen. Ein „minimaler Baukasten“ an Sprachen zur Entwicklung von Objektsystemen, auf die man nicht verzichten kann, besteht dann noch aus den *Klassendiagrammen* und den *Zustandsdiagrammen* (siehe [Harel und Gery 1997]).[1] Klassendiagramme dienen der Beschreibung der möglichen Strukturen, Zustandsdiagramme der Beschreibung des

[1] Diese Aussage ist nicht präzise, da sie nicht im formalen Sinne (z.B. im Sinne der Turing-Mächtigkeit) bewiesen wird und sich auf eine nicht ganz UML-konforme Version der Zustandsdiagramme bezieht. Kritik an UML, die vor allem die nicht eindeutig definierte Semantik betrifft, wird hier außer acht gelassen. Es sei auf die einschlägigen Tagungen (z.B. ECOOP, siehe [Kilov und Rumpe 1997]) und Veröffentlichungen (siehe z.B. [Breu et al. 1997]) zu diesem Thema hingewiesen.

dynamischen Verhaltens einzelner Objekte und durch ihr gleichzeitig definiertes Zusammenspiel auch des Verhaltens des gesamten Objektsystems.

Aus Sicht der in dieser Arbeit betrachteten komponentenbasierten Softwareentwicklung sind diese beiden Sprachen (Klassen- und Zustandsdiagramme) essentiell. Zusätzlich wird jedoch ein Sprachmittel zur Beschreibung von Komponenten benötigt. In UML können dazu Familien von Klassen und Interfaces als sogenannte *Packages* (*Pakete*) definiert werden. Die dafür zur Verfügung stehenden *Paketdiagramme* sind aus dem Blickwinkel eines „minimalen Baukastens" verzichtbar, unterstützen jedoch maßgeblich die Strukturierung größerer Systeme und damit die Wiederverwendung ihrer Subsysteme. Die Pakete stellen sozusagen die elementaren Einheiten der Wiederverwendung – im Sinne der vorliegenden Arbeit die Komponenten[1] – dar.

Es sollen im Folgenden die angesprochenen drei Diagrammtypen genauer vorgestellt werden. Die Zustandsdiagramme werden etwas ausführlicher präsentiert, da sie besonders auf die Verhaltensbeschreibung abzielen.

4.3.1 Klassendiagramme

Aus konzeptueller Sicht stellen Klassen Typen von Gegenständen[2] des betrachteten Weltausschnitts dar. Klassen werden durch ihre Eigenschaften (Attribute) und Fähigkeiten (Operationen/Prozeduren/Methoden) näher beschrieben. Objekte einer Klasse können Beziehungen zu Objekten anderer Klassen haben, die in Klassendiagrammen als Beziehungstypen bezeichnet werden. Es gibt mehrere Arten von Beziehungstypen: Assoziationen, Aggregationen, Kompositionen und Generalisierungen. Man spricht von Assoziationen zwischen Klassen, wenn die Objekte zweier Klassen „irgendwie", beispielsweise funktional, in Beziehung stehen. Eine Assoziation ist eine Aggregation, wenn

---

[1] In UML gibt es ein spezielles Sprachelement „Component". Die Abgrenzung zwischen den Begriffen „Package" (oder spezieller: „Subsystem") auf der einen und „Component" auf der anderen Seite ist alles andere als klar. In [Fowler und Scott 1998] werden diese Begriff sogar synonym interpretiert. In der UML-Spezifikation [Booch, Rumbaugh und Jacobson 1997b] sollen die Begriffe anscheinend der Unterscheidung im Sinne von Typ und Instanz dienen: Ein Package entspricht also einer Komponente gemäß der Definition in Abschnitt 3.2.1, während eine Component einer Instanz eines Paketes, also einer Komponenteninstanz (siehe Abschnitt 3.2.9), entspricht.

[2] Der Begriff „Gegenstand" umfaßt hier nicht nur Dinge, sondern auch Geschehnisse (vgl. 5.3). Das gleiche gilt für den Klassenbegriff in Programmiersprachen wie Java. Im Gegensatz zu der oft vorherrschenden Meinung, Klassen würden nur statische Aspekte repräsentieren, verkörpern sie oft auch dynamische Aspekte, wie z.B. Kommandos, Prozesse oder Strategien (siehe [Gamma et al. 1995]).

der Beziehungstyp eine *mereologische* Struktur, also eine Teile-Ganzes-Beziehung, beschreibt. Eine Komposition ist eine spezielle Aggregation, bei der ein Teil integraler Bestandteil seines übergeordneten Objekts (Ganzes) ist. Die Existenz eines Teils ist dabei unmittelbar an die Existenz des Ganzen gebunden. Alle Assoziationen können wie auch in der konzeptuellen Datenmodellierung mit Kardinalitäten versehen werden. Neben den Assoziationen gibt es noch die Generalisierung, die einen taxonomischen Beziehungstyp zwischen einer speziellen und einer generellen Klasse darstellt. Die Objekte der spezielleren Klasse können Objekte der generellen Klasse substituieren und haben u.U. weitere Eigenschaften und Fähigkeiten. Eine spezielle Klasse kann mehrere generelle Klassen (*Mehrfachvererbung*) haben und umgekehrt.

Aus Entwurfssicht können noch Interfaces von Klassen unterschieden werden, die nur Signaturen der Prozeduren, aber keine Implementierung beinhalten.

4.3.2 Zustandsdiagramme

Durch ein Zustandsdiagramm wird das Verhalten der Objekte einer Klasse definiert. Ein UML-Zustandsdiagramm beschreibt eine UML-Zustandsmaschine, deren Bestandteile hier zuerst abstrakt und danach konkreter anhand von Beispielen beschrieben werden. Zum Schluss dieses Abschnitts wird die Semantik der Zustandsmaschine umgangssprachlich beschrieben.

Die Definition der Zustandsdiagramme soll hier im Wesentlichen aus der oben zitierten Originalliteratur entnommen werden. In einigen Fällen wird von der Originaldefinition abgewichen, da sie zurzeit noch nicht vollständig und vor allem nicht eindeutig ist (siehe [Breu et al. 1997]) und außerdem nicht alle Details in Hinblick auf die hier benötigten Anforderungen abgedeckt werden.

Der Ausgangspunkt für ein UML-Zustandsdiagramm ist die State-Machine (Zustandsmaschine), die eine abstrakte Maschine darstellt. Eine State-Machine besteht im Wesentlichen aus *Zustandsknoten* (StateVertex), die entweder primitive Knoten (Pseudostates, d.h. Start-, End-, Fork-, Join-Knoten) oder normale Knoten (State) sein können, und aus Transitionen. Zustände können komplexe (Composite State) oder einfache Zustände (Simple State) sein. Komplexe Zustände können wiederum parallele oder alternative Zustände sein, in denen ihrerseits wieder andere (sowohl einfache sowie komplexe) Zustände enthalten sein können.

Transitionen verbinden zwei Zustandsknoten miteinander. Sie bestehen aus einer Ereignisdefinition (Trigger), einem Wächter (Guard) und einer Sequenz von Aktionen (ActionSequence). Ein solches Tripel wird auch als *ECA-Regel* (Event, Condition, Action) bezeichnet. Eine Ereignisdefinition besteht aus einem Ereignisnamen und einer Liste von Argumenten. Ein Wächter ist ein boolescher Ausdruck, eine Sequenz von Aktionen besteht

aus einer Sequenz von Prozedur-/Methodenaufrufen und einer Sequenz von Sendebefehlen (Signalereignissen, s.u.).

Die Notation (konkrete Syntax) der UML-Zustandsdiagramme soll im Folgenden an Beispielen verdeutlicht werden.

Zustände (siehe Abbildung 4.5) werden als Rechtecke mit abgerundeten Ecken und Transitionen als Kanten zwischen Zuständen dargestellt. Eine Transition hat zusätzlich eine Beschriftung, in der ihre Bestandteile (Trigger usw.) definiert werden.

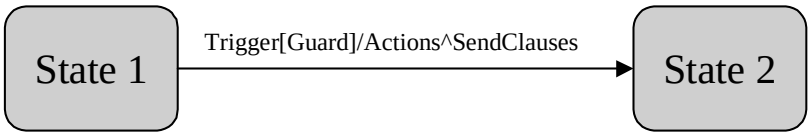


Abbildung 4.5: Zwei einfache Zustände mit einer Transition

Zustände können hierarchisch aufgebaut werden. Dazu werden parallele komplexe Zustände durch gestrichelte Linien innerhalb ihres umgebenen Zustandes getrennt dargestellt. Im Beispiel (siehe Abbildung 4.6) besteht der Zustand „State N“ aus zwei anonymen Unterzuständen (Regionen). Jeder dieser anonymen Unterzustände stellt wiederum einen komplexen Zustand dar. Sowohl der linke als auch der rechte Unterzustand sind sequentiell. Neben einfachen Zuständen enthalten diese beiden komplexen Zustände noch je zwei Pseudozustände – einen Start- und einen Endzustand.

Es ist sinnvoll, die ursprüngliche Syntax einzuschränken, indem nur Transitionen zugelassen werden (siehe z.B. [Levi 1996]), die innerhalb einer Ebene eines komplexen Zustandes verlaufen und so von der Originaldefinition für Statecharts (siehe [Harel und Naamad 1996]) abweichen. So ist z.B. eine Transition von „State 1“ in den Unterzustand „Sub State 3“ im Gegensatz zur Originaldefinition hier nicht erlaubt. Ferner werden komplexe Transitionen (mit Fork- und Join-Zuständen) sowie sogenannten History-Zustände ignoriert.

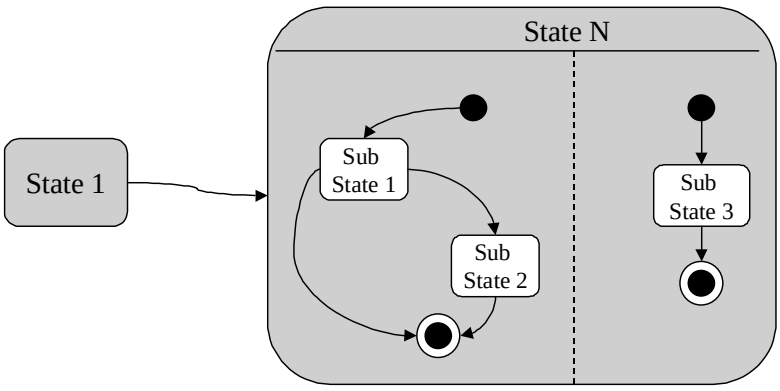


Abbildung 4.6: Ein komplexer Zustand

Die Semantik der Zustandsmaschine wird hier nur umgangssprachlich beschrieben. In UML wird das Verhalten eines Objekts einer Klasse durch ein der Klasse zugeordnetes, evtl. hierarchisches Zustandsdiagramm definiert. Das Gesamtverhalten eines Objektsystems wird dann durch die Verhaltensbeschreibungen aller ihrer Klassen bestimmt, die durch ein Klassendiagramm zusammengesetzt werden.[1] Damit die Objekte nicht isoliert nebeneinander her leben, können Objekte einander Nachrichten (An-, Aufforderungen, engl. Requests) schicken, die als sogenannte *Ereignisse* auf der Empfängerseite ankommen.

Die *Semantik* durch Zustandsdiagramme definierter Zustandsmaschinen basiert im Wesentlichen auf der Verarbeitung von Ereignissen (Events). Von einem Objekt erzeugte Requests aktivieren Transitionen eines anderen Objekts, wenn die Signatur des Requests zu dem Trigger (= Eventname mit einer Liste von Variablen) einer Transition passt und zusätzlich ein boolescher Ausdruck (Guard) der Transition erfüllt ist. Wenn eine Transition aktiviert ist, wird sie zusammen mit einer konfliktfreien Menge anderer Transitionen zum „Feuern“ ausgewählt. Eine solche konfliktfreie Menge von Transitionen ist eine Menge, in der nur Transitionen vorkommen, die nicht das gleiche Objekt verändern oder auf das gleiche Ereignis reagieren wollen. Wurde eine Transition ausgewählt, so übernimmt sie die mit dem Ereignis mitgelieferten Informationen und führt eine Sequenz von Aktivitäten (ActionSequence) aus. Die Aktivitäten können wiederum neue Ereignisse erzeugen, die andere Transitionen aktivieren usw. Nach der Ausführung einer Transition wechselt das Objekt den Zustand vom Start- in den Zielzustand der Transition.

Die konfliktfreie Menge von ausgewählten Transitionen muss immer maximal sein. D.h. jede Transition, die feuern kann und nicht im Konflikt mit einer Transition dieser Menge steht, muss in die Menge aufgenommen werden. Diese maximale Menge ist nicht notwendigerweise eindeutig. Im mehrdeutigen Fall wird per Zufall irgendeine gültige Menge ausgewählt. Somit können Zustandsmaschinen auch nicht-deterministisch sein.

Transitionen, deren Zielzustand ein komplexer Zustand ist, verhalten sich so, dass die parallelen internen Startzustände (schwarze Punkte in Abbildung 4.6) aktiviert werden und von dort sofort zu den ersten „normalen“ Folgezuständen gewechselt wird. Von dort erfolgt die weitere Verarbeitung der Ereignisse, bis alle Zielzustände erreicht werden. Der Fall, dass der komplexe Zustand nur alternative Zustände enthält, ist offensichtlich ein Sonderfall eines komplexen Zustandes mit nur einem parallelen Subzustand.

Die Semantik ist beschränkt durch eine sogenannte „Run-to-completion“-Vorgehensweise, die dafür sorgt, dass eine Zustandsmaschine eines Objektes die Ereignisse sequentiell und erst

[1] In der original UML-Spezifikation wird ignoriert, dass ein „erstes“, initiales Objekt existieren muss, das alle anderen Objekte erzeugt.

dann verarbeitet, wenn die Maschine wieder in eine stabile Konfiguration gelangt ist (Stichwort: Thread Protection). Eine Konfiguration ist dabei die Menge aller augenblicklich aktiven Zustände einer Maschine. Eine stabile Konfiguration wird dann erreicht, wenn alle Transitionen einer konfliktfreien Menge verarbeitet wurden und auch alle von diesen Zuständen folgenden „completion transitions“ (Transitionen, die keinen Trigger besitzen) ausgeführt wurden[1]. Denkbare Konfigurationen der Zustandsmaschine in Abbildung 4.6 sind z.B. {State1}, {StateN, SubState1, SubState3} und {StateN, SubState2, SubState3}.

Aktionen der Aktionsfolge können sowohl synchron als auch asynchron abgearbeitet werden. Bei asynchroner Ausführung setzt die ausführende Transition einen Request ab und wartet (im Gegensatz zur synchronen Ausführung) nicht darauf, dass die empfangende Zustandsmaschine diesen Request abgearbeitet hat. Der Unterschied ist, dass Änderungen der Eigenschaften des sendenden Objektes, die durch einen Request entstehen, nur bei der synchronen Bearbeitung von Aktionsfolgen garantiert im nächsten Schritt schon vorliegen.

4.3.3 Paketdiagramme

In Paketdiagrammen werden *Pakete* als Komposita von Modellierungselementen (Klassen, Interfaces, Assoziationen, Subpakete usw.) aufgefasst und stellen gleichzeitig einen lokalen Namensraum für diese Elemente dar. Außerdem kann ein Paket Elemente anderer Pakete referenzieren. Es wird dann gesagt, dass ein Paket ein anderes Paket *importiert*. Die Tatsache der Referenzierung wird durch eine Importbeziehung explizit dokumentiert.

Das Verhalten eines Paketes ergibt sich direkt aus dem Verhalten der ihnen zugrunde liegenden Klassen und der Klassen aller (auch tiefer geschachtelten) Subpakete usw. Zur abstrakteren Beschreibung des dynamischen Verhaltens eines Paketes können in UML zusätzlich Kollaborationsdiagramme eingesetzt werden. Diese werden dann den Paketen zugeordnet. Jedoch ergibt sich bei der Komposition von Paketen nicht automatisch das Verhalten des neuen Systems aus den Kollaborationsdiagrammen der Subpakete, d.h., dass die Kollaborationsdiagramme pro Paket manuell definiert werden müssen. Hier ist eine Automatisierungsunterstützung denkbar, die heutige Werkzeuge jedoch noch nicht bieten.

Ferner sind Pakete generalisierbar. Die Bedeutung dieser Generalisierung ist in UML nicht eindeutig geklärt. Generalisierung zwischen Paketen könnte beispielsweise bedeuten, dass ein generelles Paket durch alle seine Spezialisierungen substituierbar ist.

---

[1] Ein Problem dieser Vorgehensweise ist, dass Objekte keine Rekursionen verarbeiten können, denn ein in der Rekursion erzeugtes Request könnte erst nach der Verarbeitung des verursachenden Ereignisses verarbeitet werden. Dies wird aber durch die Forderungen des "run-to-completion" verhindert.

4.4 Sprachen zur Beschreibung von Softwarearchitekturen

Architekturbeschreibungssprachen (Architecture Description Languages, ADLs) werden ähnlich wie die zuvor vorgestellten Sprachen zur Softwareentwicklung (siehe Abschnitt 4.3) – allerdings hier ausschließlich für den wesentlich gröberen Architekturentwurf – eingesetzt. So unterscheiden sich die betrachteten Aspekte und die verwendeten Sprachparadigmen stark voneinander. Während bei UML die verarbeitenden und zu verarbeitenden Objekte im Vordergrund stehen, wird bei den ADLs unabhängig vom zugrunde liegenden Programmierparadigma die Komponentenorientierung betont. Es werden also größere Softwareelemente – eben die Komponenten – und ihr Zusammenspiel beschrieben, wobei von der detaillierten Betrachtung des Verhaltens einer Komponente abgesehen wird. Die ADLs konzentrieren sich dabei auf die Aspekte des Zusammensetzens (*Glueing*) der Softwarekomponenten. Im Folgenden wird die Sprache *Darwin* als ein repräsentativer Vertreter dieser Klasse von Sprachen vorgestellt.

4.4.1 Darwin

Darwin (siehe [Magee et al. 1995]) ist eine Architekturbeschreibungssprache, die einen komponentenbasierten Ansatz zur Strukturierung von Softwaresystemen unterstützt. Eine Komponente ist das Grundelement der Struktur. Sie „versteckt“ ihr Verhalten hinter einer wohldefinierten (direkten) Schnittstelle. Schnittstellen definieren die Abhängigkeit der Komponente von ihrer Umwelt (d.h. von andere Komponenten im System) und stellen Dienste dar, die eine Komponente anbietet oder benötigt. Programme werden konstruiert, indem man Instanzen von Komponenten(typen) erzeugt und deren Schnittstellen verbindet. Komponenten können selbst wieder eine Substruktur aus anderen Komponenten besitzen. Die allgemeine Struktur eines Programms in Darwin ist folglich ein Baum, in dem die Zwischenknoten zusammengesetzte Komponenten und die Blätter atomare Komponente sind. Eine atomare Komponente hat keine Substruktur und definiert statt dessen ein Verhalten. Das Verhalten einer zusammengesetzten Komponente definiert sich über das Verhalten der Subkomponenten. Eine zusammengesetzte Komponente kapselt alle Schnittstellen zwischen ihren Subkomponenten, die nicht an eine externe Schnittstelle dieser Komponente angeschlossen sind. Darwin unterstützt neben der Unterscheidung von Struktur und Verhalten auch zwei verschiedene Sichten auf das Verhalten: die *Behavioural View* definiert die generellen Anforderungen an die Komponente (Analyse) und die *Service View* definiert die konkrete Realisierung der Komponente im Rahmen der Konstruktion (Entwurf). Aus Sicht der Anforderung wird nur definiert, welche Aktionen von zwei kommunizierenden Komponenten ausgeführt werden sollen. In der zweiten Sicht wird dann auch definiert, welche Komponente welche Teile der Implementierung enthält.

Darwin bietet eine textuelle und eine graphische Sprache, die auf Basis entsprechender Werkzeuge zur Definition von Softwarearchitekturen genutzt werden können. Der Software-Architekturassistent (SAA) ist ein graphisches Werkzeug für den Entwurf und die Entwicklung von verteilten Programmen auf Basis von Darwin-Architekturbeschreibungen. Aus dieser Umgebung können Programme generiert werden, die vom Darwin-Compiler zu lauffähigen Programmen übersetzt werden.

Darwin wird zur Spezifikation der Struktur und zum darauf folgenden schrittweisen Aufbau verteilter Systeme genutzt. Ähnlich können Softwarearchitekturen auf Basis von Darwin zur Systemmodellierung und -analyse verwendet werden.

4.4.2 Semantik von Darwin

Die Semantik von Darwin wird durch eine Übersetzersemantik in den π-Kalkül (zum Kalkül siehe Kapitel 17) definiert. Im Folgenden sollen die Ideen, die dieser Übersetzung zugrunde liegen, an einfachen Beispielen erklärt werden. Graphisch wird in Darwin eine Komponente als Rechteck (siehe Abbildung 4.7) dargestellt. Die eingehenden Schnittstellen einer Komponente werden als weiße, die ausgehenden Schnittstellen als schwarze Punkte dargestellt. In der textuellen Definition werden die eingehenden Schnittstellen mit dem Schlüsselwort `require` und die ausgehenden Schnittstellen mit dem Schlüsselwort `provide` gekennzeichnet. Verbindungen zwischen Komponenten – in diesem ersten Beispiel sind keine Verbindungen sichtbar – werden durch das Schlüsselwort `bind` definiert.

Um nun das Komponentenmodell, das Darwin zugrunde liegt, formal zu definieren, werden die vorgestellten Konstrukte (`require`, `provide` und `bind`) in den π-Kalkül abgebildet.

Eine ausgehende Schnittstelle wird durch den π-Kalkül-Agenten `PROV` repräsentiert, der einen Service `s` beliebig oft („!“) anbietet. Eine Instanz `p` der Schnittstelle `PROV` wird durch den Agenten `PROV(p,s)` erzeugt. Im π-Kalkül wird `PROV` folgendermaßen definiert[1]:

```
PROV(p,s) = !(p?x.x!s)
```

Auf dem Kanal `p` wartet der Prozess auf Mitteilung eines Kanals `x`, auf der danach der Service `s` vermittelt wird. Der Service `s` wird nicht weiter definiert, da er auf der softwarearchitektonischen Ebene nicht von Interesse ist. Es wird einfach nur vorausgesetzt, dass er unter der „Adresse“ s abrufbar ist. Objekte eines objektorientierten Systems werden somit nur als abstrakte Services betrachtet, über die keine weiteren Aussagen gemacht werden können und sollen.

---

[1] Hier wird nicht die original π-Kalkül-Syntax, sondern eine an Pict (siehe Kapitel 17) angelehnte Syntax gewählt.

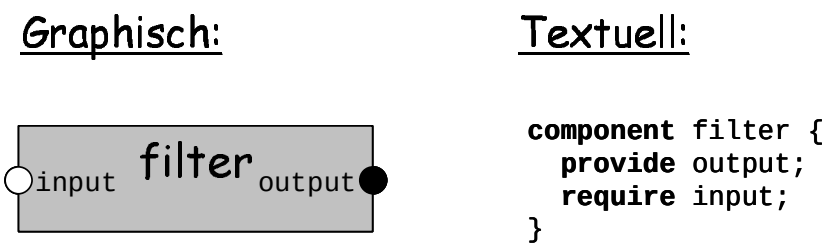


Abbildung 4.7: Einfache Darwin-Komponente

Eine eingehende Schnittstelle wird durch den Agenten `REQ` repräsentiert, der einen Service einmal an einer vom ihm verwalteten Stelle `l` benötigt. Eine Instanz `r` des Komponententyps `REQ` wird durch den Agenten `REQ(r,l)` erzeugt. Im π-Kalkül wird `REQ` folgendermaßen definiert:

```
REQ(r,l) = r?y.y!l
```

Auf dem Kanal `r` wartet der Prozess auf die Mitteilung des Kanals `y`, über den danach die Stelle `l` referenziert wird. Was die Stelle `l` mit dem Service macht, wird ebenfalls nicht betrachtet.

Zur Verbindung zweier Komponenten stellt Darwin ein Verbindungskonstrukt `bind` zur Verfügung, das ebenfalls als Agent im π-Kalkül (`BIND`) realisiert wird. Es verbindet eine Komponente `x`, die einen Service zur Verfügung stellt, mit einer Komponente `y`, die diesen Service benötigt. Dazu wird der anfordernden Schnittstelle `r` einfach der Name der bereitstellenden Schnittstelle `p` zugeschickt.

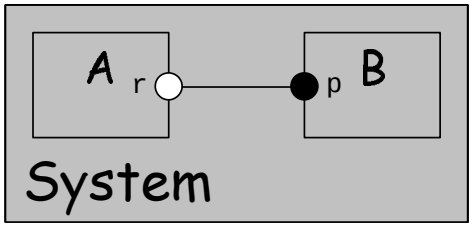
```
BIND(r,p) = r!p
```

Nun muss definiert werden, wie eine komplette Komponente `X` im π-Kalkül abgebildet wird. Dazu wird die Komponente als ein Agent `AgentX` abgebildet, der aus einer parallelen Komposition der Agenten der ein- und ausgehenden Schnittstellen sowie einem (oder mehreren) Agenten zur benutzerdefinierten Implementierung des Verhaltens der Komponente (`BehaviourX`) besteht. Allgemein lässt sich ein solcher Agent durch folgendes Schema veranschaulichen:

$$
\begin{aligned}
&\texttt{AgentX}(p_1,\ldots,p_n,r_1,\ldots,r_m) = \\
&\quad (\texttt{new } s_1..s_n\ l_1...l_m)\ (\quad \texttt{PROV}(p_1,s_1) \mid \ldots \mid \texttt{PROV}(p_n,s_n) \mid \\
&\qquad\qquad\qquad\qquad\quad \texttt{REQ}(r_1,l_1) \mid \ldots \mid \texttt{REQ}(r_m,l_m) \mid \\
&\qquad\qquad\qquad\qquad\quad \texttt{BehaviourX}(s_1,\ldots,s_n,l_1,\ldots,l_m) \quad)
\end{aligned}
$$

Die eingehenden Schnittstellen werden durch die Namen $p_1,\ldots,p_n$ und die ausgehenden Schnittstellen durch die Namen $r_1,\ldots,r_m$ repräsentiert. Zur Veranschaulichung der Übersetzung einer komplexeren Komponente dient die Komponente `System` (siehe Abbildung 4.8).

Graphisch: Textuell:




```
component Server {
  provide p;
}

component Client {
  require r;
}
```

```
component System {
inst      A:Client;
          B:Server;
bind
  A.r -- B.p
}
```

Abbildung 4.8: Komplexe Darwin-Komponente

Die Subkomponenten `Server` und `Client` werden nach dem vorgestellten Schema in die folgenden Agenten übersetzt:

```
AgentServer(p) = (new s) (PROV(p, s) | BehaviourServer(s))
AgentClient(r) = (new l) ( REQ(r, l) | BehaviourClient(l))
```

Eine komplexe Darwin-Komponente wird nun durch die parallele Komposition ihrer Subkomponenten und der Agenten für die angegebenen Verbindungen in den π-Kalkül übersetzt. Für die Komponente `System` würde dann folgender Agent erzeugt :

```
AgentSystem() =
     (new r p) (AgentClient(r) | AgentServer(p) | BIND(r,p))
```

Die Komponente `System` ist eine geschlossene Komponente, die keine Schnittstellen besitzt und somit nicht mehr mit anderen Komponenten gekoppelt werden kann. Es bleibt also noch zu klären, wie die Schnittstellen einer komplexen Komponente zur Außenwelt definiert werden können. Dazu können Verbindungen zwischen Schnittstellen dieser komplexen Komponente mit noch nicht besetzten Schnittstellen von Subkomponenten erstellt werden. Hätte beispielsweise die Komponente `Server` eine zusätzliche ausgehende Schnittstelle (`provide x;`), auf der sie einen weiteren Service `s'` anbietet, so könnte die komplexe Komponente `System` diesen Service über eine zusätzliche Schnittstelle `y` verfügbar machen. Folgende textuelle Komponentendefinition wäre dann in Darwin anzugeben:

```
component System {
provide y;
     inst A:Client;
          B:Server;
     bind
          A.r -- B.p;
          y   -- B.x;
}
```

4.5 Einordnung der Sprachen zur Verhaltensbeschreibung

Anhand der Kategorien, die in Abschnitt 4.1 vorgestellt wurden, sollen nun die drei ausgewählten Sprachen (Abschnitt 4.2 bis 4.4) bewertet werden. Die folgende Tabelle stellt die Vor- und Nachteile der drei betrachteten Sprachansätze gegenüber.

Sprachen **Kriterien**	**EPK**	**UML-Statecharts**	**Darwin**
Paradigmen	Endlicher Automat	Endlicher Automat	Prozessterme
Substituierbarkeit	nein, keine Unterstützung	Inheritance subtyping	ja, aber keine systematische Unterstützung
Unterspezifikation	ja	nein	ja (durch zwei Beschreibungsebenen)
Ausdrucksstärke (Expressive Power)	formal: - inhaltlich: +	formal: + inhaltlich: -	formal: o inhaltlich: -
Verständlichkeit	Verständlichkeit: + Eindeutigkeit: -	Verständlichkeit: - Eindeutigkeit: o	Verständlichkeit: o Eindeutigkeit: +
Beschreibungszweck: ergebnis- versus prozessorientiert	ergebnis- und prozessorientiert	prozessorientiert	prozessorientiert
Abstraktionsbereich: geschehnis- versus zustandsorientiert	geschehnis- und zustandsorientiert	geschehnis- und zustandsorientiert	geschehnisorientiert
Strukturierung: objekt- versus vorgangsorientiert	vorgangsorientiert	objektorientiert	objektorientiert
Kompositionalität	-	o	+
Variablen	-	+	+

Tabelle 4.1: Bewertung von Sprachen zur Verhaltensbeschreibung von Komponenten

4.6 Fazit

Für eine im Rahmen dieser Arbeit zur konstruierende Sprache gilt es, möglichst die besten Eigenschaften der in diesem Kapitel dargestellten Sprachansätze zu vereinigen, ohne dadurch wesentliche Nachteile zu implizieren. Alle Ansätze werden zur Beschreibung des Verhaltens von Softwareprodukten eingesetzt. Jeder dieser Ansätze kommt in einer anderen Phase des Softwarelebenszyklus zum Einsatz. UML wird bei der eigentlichen Software-(komponenten)entwicklung, Darwin bei der Komposition bestehender Softwarekomponenten und ARIS bei der Vorbereitung der Konfiguration und zur Beschreibung (Referenzmodellierung) bestehender (monolithischer) Anwendungssoftwareprodukte eingesetzt. Eine durchgehende Methode, die auch die verschiedenen Anforderungen, der an diesem Prozess beteiligten Akteure berücksichtigt, gibt es bislang nicht und kann es vielleicht auch gar nicht geben, da sie zu komplex würde. Dennoch soll in dieser Arbeit, die den Fokus auf die anwendungsbezogene Beschreibung von Anwendungssoftwareprodukten setzt, ein Vorschlag unterbreitet werden, der möglichst früh im Lebenszyklus diesen Aspekt berücksichtigt und zeigt, wie anwendungsbezogene Beschreibungen konsequent aufgebaut werden können. Eine zu entwickelnde Sprache sollte, dann geschäftsprozessorientiert sein wie die EPK, objektorientierte Eigenschaften wie die UML besitzen und Kompositionalitäts-aspekte wie Darwin aufweisen.

5 Normsprachen

5.1 Formale versus natürliche Sprache

Die Frage nach dem Aufbau einer Sprache wird in der Informatik i.d.R. mit der Definition von Syntax und Semantik beantwortet. So werden Sprachen zur Beschreibung des Verhaltens von Systemen – um die geht es in dieser Arbeit – beispielsweise syntaktisch mittels kontextfreier Grammatiken und semantisch auf Basis von Transitionssystemen aufgebaut. Durch das Transitionssystem werden Umformungsregeln auf den durch die Syntax vorgegebenen Beschreibungen definiert, durch die u.a. Aktionen des beschriebenen Systems abgeleitet werden können. Die *formale Semantik* einer Verhaltensbeschreibung kann beispielsweise durch die Menge aller Aktionsfolgen (= Traces), die man aus einer Verhaltensbeschreibung systematisch ableiten kann, definiert werden. Sowohl die kontextfreie Grammatiken als auch Transitionssysteme können als Teilsprachen der Prädikatenlogik 1.Stufe verstanden werden.

Da Programmiersprachen der Befehlssteuerung von Rechnern dienen, ist es nicht verwunderlich, dass lange Zeit formale Aspekte bei der Sprachdefinition und der Nutzung (Programmierung) im Vordergrund standen. Erst im Laufe der Entwicklungsgeschichte der Programmiersprachen hat man mehr und mehr Energie in die Verbesserung der Verständlichkeit der Sprachen und ihrer Produkte (= Programme) gelegt. Das führte letztlich dazu, dass Entwicklungen im Bereich der objektorientierten Programmiersprachen suggerieren sollten, dass die „Abbildung der Realität“ im Rechner kaum noch ein Problem sei. Meyer (siehe [Meyer 1988]) ließ sich etwa zu der folgenden Aussage verleiten:

„... in the physical or abstract reality being modeled, the objects are just there for the picking.“

Das diese naive Vorstellung von der Softwareentwicklung nicht greift, ist jedem Softwareentwickler allzu verständlich, da es viele alternative Möglichkeiten gibt, Sachverhalte der Diskurswelt durch ein objektorientiertes Programm zu beschreiben. Im Vordergrund der Beschreibung steht die Frage, was mit dem System erreicht werden soll – in [Wedekind 1992] wird in diesem Zusammenhang von *Thematisierung* gesprochen. Gerade in der thematisierten Beschreibung der Sachverhalte und der damit notwendigen Abstraktion vom Einzelfall liegt die geistige Leistung des Entwicklers und nicht in dem bloßen „Aufpicken“ der Objekte. Was ein Objekt ist, ist nicht von vornherein gegeben, sondern hängt stark vom intendierten Zweck der zu entwickelnden Software ab.

Die Forderung nach Abstraktion vom Einzelfall und – im Sinne der in dieser Arbeit betrachteten Wiederverwendung von Softwarekomponenten – möglichst vielen Anwendungsbereichen steht im Widerspruch zur Verständlichkeit eines Softwareproduktes.

Ein Kernproblem ist, dass die formalen Sprachelemente zwar für die Softwareentwicklung ausreichen, jedoch für den „Leser“ und „Wiederverwender“ zu abstrakt sind. So ist im Nachhinein an einer Klassendefinition in einer objektorientierten Programmiersprache nicht genau zu erkennen, ob durch eine Klasse eher „Dinge“ oder „Geschehnisse“ der Diskurswelt beschrieben werden. Dieses Manko wurde in den letzten Jahren erkannt und hat zur Etablierung sogenannter Entwurfsmuster (siehe [Gamma et al. 1995]) geführt. Entwurfsmuster sind konkret(er)e Strukturen von Objektfamilien, die immer wieder in Problemstellungen auftauchen. So gibt es Muster zur Beschreibung von „Kommandos“, „Strategien“ und „Hierarchien“. Im Sinne der Wiederverwendung ist es wichtig, dass der Entwickler anzeigt, welche Muster er benutzt hat, damit der „Wiederverwender“ eine Komponente schneller versteht.

Ein Problem der „Musterfraktion“ ist, dass die Muster nicht konstruktiv hergeleitet wurden, sondern eher als Erfahrungsberichte erfahrener Softwareentwickler und Systemanalysten zu verstehen sind. Daraus ergeben sich offenbar die Nachteile, dass es keine klare Abgrenzung zwischen den Erfahrungsmustern verschiedener Entwickler gibt und dass diese Muster auch erst einmal wieder gelernt werden müssen.

Wesentlich für die Verständlichkeit und damit für die Wiederverwendung ist nicht nur die normierte Definition von Klassen- und Objektstrukturen, sondern auch der Methodenaufrufe, also des dynamischen Teils eines objektorientierten Systems. Bislang gibt es keinerlei Normen oder Entwurfsmuster, die den Aufbau der Signaturen von Methoden standardisieren. Anhand der Signatur kann meist nicht erkannt werden, in welchem Verhältnis die Argumente zueinander stehen. Für das Verständnis ist z.B. wesentlich, welche Argumente die Hauptelemente und welche Argumente z.B. nur Eigenschaften angeben, also nebensächlich (aber nichtsdestotrotz wichtig) sind. Als kleines Beispiel sei ein Druckmanager genannt, der als Klasse in einer Programmiersprache wie folgt angegeben sei:

```
class printerManager {
     Boolean check (Printer: p, Document: d, Person: u) {…}
     …
}
```

Aus der Signatur der Methode `check` kann nun nicht unmittelbar abgelesen werden, welche(n) Aussagentypen diese Methode beschreibt. Viele Interpretationen sind denkbar:

1. Es wird geprüft, ob ein Dokument `d` auf einem Drucker `p` ausgedruckt werden darf, damit eine Person `u` es zum Lesen bekommt.

2. Es wird geprüft, ob ein Benutzer `u` ein Dokument `d` auf dem Drucker `p` ausdrucken darf.

3. Es wird geprüft, ob ein Dokument `d` zusammen mit den Daten der Person `u` (z.B. Absender) auf dem Drucker `p` ausgedruckt werden kann/darf.

Um also verstehen zu können, was eine Methode bedeutet, sind entweder zusätzliche Beschreibungen heranzuziehen, oder der „Leser“ muss sich den Programmcode genauer ansehen. Dieses Ansehen des Programmcodes führt dann soweit, bis der Leser auf Klassen und Methoden bekannter Softwarepakete/-bibliotheken stößt. Zusätzliche Beschreibungen, die sicherlich die bessere Alternative darstellen, dürfen aber nicht statisch sein, sondern müssen sich entsprechend der Parametrierung einer Komponente und ihres Einsatzfeldes möglichst einfach anpassen lassen.

Durch die Formalisierung des ursprünglichen Aussagentyps werden in den Signaturen im Rahmen der Entwicklung Informationen „wegabstrahiert“, die für die Programmausführung unwichtig, für die Wiederverwendung jedoch wesentlich sind. Die tatsächliche Bedeutung einer Methode kann also nur durch Lesen des gesamten Programmtextes oder zusätzlich ausgelieferter Beschreibungen verstanden werden. Hinzu kommt, dass man aus rein formaler Perspektive die Menge der eingeführten Symbole (`„Printer“`, `„Person“`, `„Document“`) durch einen Isomorphismus auf eine andere Menge von Symbolen – beispielsweise `„XYZ“`, `„Auto“` und `„Paper“` – abbilden kann, ohne dass die Korrektheit oder die Qualität eines Programms darunter „leiden“ würde. Jedoch würden dadurch die Lesbarkeit und das Verständnis für die implizierte Semantik weiter reduziert. Da aber die Wahl der Symbole und der gesamten Syntax durch unterschiedlichste Kriterien bestimmt wird, sind solche Programme auch bei „guter“ Wahl der Symbole oft nur Experten zugänglich.

Zwei Ansätze der Interpretation lassen sich also unterscheiden: Bei der formal-logischen Interpretation werden den Symbolen der Beschreibung explizit Mengen (Extensionen) zugeordnet. Bei der semi-formalen Interpretation haben die Symbole per se eine Bedeutung, sind also Prädikatoren. Bei dem zweiten Ansatz ist fraglich, ob die Bedeutung für jeden Leser die gleiche ist.

Wie angedeutet können Softwareprodukte auch natürlichsprachlich beschrieben werden, und so wird selbst zur Erklärung formaler Systeme fast immer auf die natürliche Sprache zurückgegriffen. Sie dient aber hauptsächlich der Erläuterung und nicht der eigentlichen Definition[1]. Die natürliche Sprache wird nicht direkt zur Programmierung oder zur Beschreibung von Systemen genutzt, da die Sätze oft nur in einem übergeordneten Zusammenhang (Kontext) verstanden werden können oder nicht eindeutig sind. Mehrdeutigkeiten können z.B. durch die Verwendung von Homonymen – dies sind Wörter mit verschiedenen Bedeutungen – entstehen. Synonyme können ebenfalls zu Problemen führen, wenn sie nicht kontrolliert eingesetzt werden.

---

[1] Methodisch wird letztlich immer auf eine natürliche Sprache zurückgegriffen, da als Grundlage für eine formale Sprache irgendwann eine natürliche Sprache gedient haben muss.

Obwohl es in den analytischen Sprachwissenschaften (siehe [Hartmann 1990]) Bemühungen gibt, die Bedeutungen natürlichsprachlicher Sätze methodisch zu erfassen, sind diese bislang nicht von durchschlagenem Erfolg. Beweis dafür sind automatische Übersetzungsprogramme, die zwar wertvolle Dienste leisten können, jedoch nicht ohne nachträgliches Überarbeiten der übersetzten Texte auskommen. Die (rationalen) Konstruktivisten gehen sogar davon aus, dass der forschende Grammatiker aus der natürlichen Sprache nicht alle methodischen Abhängigkeiten ihrer Mittel analysieren kann. Ja selbst die Frage, ob alle grammatischen Unterscheidungen methodisch gewinnbar sind, scheint offen zu sein.

Genau genommen gibt es somit keine geeignete, direkt verfügbare, d.h. eindeutige, aber dennoch verständliche Sprache, mit der das Verhalten von Softwareprodukten für den Anwender beschrieben werden kann. Die eine – die formale – Klasse von Sprachen ist für den Anwender nicht verständlich genug, die andere – die materiale – Klasse von Sprachen ist nicht eindeutig. Beide Sprachklassen sind also – zumindest isoliert – zur Beschreibung von Anwendungssachverhalten schlecht geeignet.

Die konstruktive Wissenschaftstheorie empfiehlt daher eine anderes Herangehen an den Aufbau einer Sprache. Sie geht davon aus, dass man nicht vollkommen losgelöst von bestehenden natürlichen Sprachen eine neue Sprache aufbauen kann und verlangt deswegen, dass die natürliche Sprache rekonstruiert und dabei „um Defekte bereinigt“ wird. Eine solche *rationale Sprache* ist somit nicht Ergebnis einer historischen Entwicklung sondern, wird mit „Absicht und Plan entwickelt“. Sie ist „... Schritt für Schritt methodisch zu konstruieren (und nicht etwa schlicht zu ‚geben‘)“ [Hartmann 1990]. Wichtigstes Merkmal der *rationalen Grammatik* ist, dass methodische Unterschiede explizit gemacht werden und nicht etwa komplexer Interpretationen bedürfen. Verbindet man eine rationale Grammatik mit einem *Lexikon*, in dem der Wortschatz einer Sprache organisiert wird, dann wird auch von einer Orthosprache (siehe [Lorenzen 1987]) gesprochen. In Anlehnung an [Schienmann 1997] nennt man solche Sprachen *Normsprachen*. In den Arbeiten von Schienmann und anderen Autoren werden auf diesem Fundament Vorschläge zum Aufbau komplexer (Fach-)Sprachen gemacht. Sie gehen von einer Gegenstandsaufteilung aus, die man anfangs *empraktisch*, d.h. parallel zu nicht-sprachlichem Handeln, erlernt und später *epipraktisch*, d.h. durch Reden über das Handeln, ausbaut. Nicht-sprachliche Handlungen der empraktischen Reden können z.B. Handlungen des Vormachens sein. Epipraktisches Reden dient dann z.B. dem Erlernen von Begriffen auf Basis von Sprachhandlungen wie Erklärungen und Definitionen.

Eine zentrale These des konstruktiven Ansatzes ist, dass eine „vernünftige“ Sprache letztlich auf empraktisches (und damit eindeutiges) Reden zurückgeführt werden kann. Mehrdeutigkeiten in einer existierenden Begriffswelt können also aufgelöst werden, wenn man die zugrunde liegende Sprache im Extremfall auf diese empraktischen Anfänge zurückführt. Dieser Prozess wird dann auch *Rekonstruktion* genannt.

5.2 Einschub: Interpretation des Customizing-Prozesses

Hier soll kurz ein offensichtlicher Zusammenhang zwischen (intensionaler) Logik und Anwendungssystemen (am Beispiel R/3) in einem übertragenden Sinne verdeutlicht werden.

Das R/3-System ist ein konfigurierbares betriebliches Anwendungssoftwareprodukt. Es kann in diesem Sinne als ein Kalkül – also als eine „logische Sprache“, in der bestimmte Probleme „leicht“ zu formulieren sind – verstanden werden, mit dem im Rahmen der Konfiguration verschiedene Aufgaben formuliert und zur Laufzeit durchgeführt werden können. Die *intensionale Semantik* des R/3-Systems ist dann die Bedeutung, die sich aus allen Einsatzmöglichkeiten des Systems ergibt. Durch den Kalkül – der das System darstellt – werden eine Reihe (tatsächlich: unendliche viele) von Interpretationen zugelassen. Diese Interpretationen können als konkrete Anwendungsbereiche (Carnap würde sagen: „als mögliche Welten“, siehe [Kutschera 1976]) verstanden werden, die durch das System unterstützt werden.

Für einen neuen Anwender stellt sich nun die Frage, wie er das System zu interpretieren hat, damit es den Anforderungen seines Unternehmens weitgehend gerecht wird. Er hat also eine – für sein Unternehmen geeignete – Interpretation anzugeben und zu prüfen, ob sie Element der Menge aller gültigen Interpretationen des Anwendungssystems ist – also auch tatsächlich durch den Kalkül des Systems unterstützt wird.

In der Logik wird eine Interpretation angegeben, indem man eine Abbildung Φ definiert, die alle Elemente des Kalküls (also des R/3-Systems) – zu diesen Elementen gehören im Wesentlichen die Symbole, die für informatorische sowie organisatorischen Einheiten und für Verfahren (Transaktionen) stehen – auf Elemente des Anwendungsbereiches abbildet. Da man die Mengen nicht direkt angeben kann, werden Begriffe gewählt, die diese Mengen referenzieren.

Das Kalkül des R/3-Systems ist so komplex, dass das System nicht unmittelbar verstanden werden kann und zur Beurteilung der Gültigkeit einer Interpretation herangezogen werden kann. Das hieße letztlich, Programmcode zu lesen und zu verstehen. Auch eine rein formale Beschreibung aller möglichen Interpretationen kann nicht als Kriterium herangezogen werden, da diese ähnlich komplex ausfallen würde. Jedoch kann man sich formale Hilfsmittel vorstellen, die zumindest helfen können, dies zu überprüfen. Und so werden ja auch bei der Eingabe der Customizing-Daten Plausibilitätsprüfungen durchgeführt, die prüfen, ob die Interpretation gültig ist. Jedoch gibt es immer wieder Fälle, in denen trotzdem fehlerhafte (d.h. ungültige) Interpretationen eingegeben werden.

Ein Hauptproblem jedoch ist nicht immer nur die reine Erstellung der Interpretation, sondern das Problem, erst einmal zu verstehen, was gültige Interpretationen sind. Dazu können

Beispielinterpretationen (vgl. SAP-IDES in [Appelrath und Ritter 1999]), partielle Interpretationen (vorkonfigurierte Systeme) oder Beschreibungen von Interpretationen, sogenannte Referenzmodelle, angegeben werden.

Referenzmodelle können somit – da sie ja vom konkreten Anwendungssystem abstrahieren – als im Vergleich zum R/3-System bzw. im Vergleich zu dem Kalkül unterspezifizierte Kalküle verstanden werden. Auch im Referenzmodell muss eine Interpretation gefunden werden.

Wichtig ist nun, dass der Prozess der Erstellung der unternehmensspezifischen Interpretation (egal ob vom Referenzmodell oder vom tatsächlichen System) ein iterativer und inkrementeller Prozess ist. Der Anwender muss nach Möglichkeit bei der Entwicklung der Abbildung Φ optimal unterstützt werden und nicht erst bei der letztlichen Beurteilung der (Un-)Gültigkeit der Interpretation.

5.3 Gegenstandseinteilung

Die im Weiteren für Normsprachen genutzte Gegenstandseinteilung wird „nicht einfach gegeben“. Die Konstruktivisten (vgl. [Lorenzen 1987], [Hartmann 1990], [Schienmann 1997]) gehen von einer Gegenstandseinteilung (siehe Abbildung 5.1) aus, die der Mensch beim Spracherwerb empraktisch erlernt und sich somit im Nachhinein rekonstruieren lässt (siehe 5.1). Da dieser Lernprozess erfahrungsgemäß sehr langwierig ist, soll hier nur das Ergebnis präsentiert werden.

Gegenstände kann man in Nicht-Geschehnisse (u.a. Dinge) und Geschehnisse unterteilen. Diese Unterscheidung ist seit Platon „klassisch“ und hat sich beispielsweise auch in der Informatik durch die Unterscheidung von *Struktur* und *Verhalten* bei der Beschreibung von Programmsystemen sowie in der betrieblichen Organisation durch die Unterscheidung von *Aufbau-* und *Ablauforganisation* durchgesetzt. Dinge lassen sich weiter in Lebewesen und „Sachen“ unterscheiden. *Geschehnisse* lassen sich in *Regungen* (veranlasst) und *Bewegungen* (nicht veranlasst) unterteilen. Regungen lassen sich weiter in *Handlungen* (beabsichtigt) und *Verhalten*[1] (nicht beabsichtigt, widerfahrend) untergliedern. Sind Handlungen partnerorientiert, dass heißt auf andere Menschen ausgerichtet, so spricht man von

[1] Der Begriff „Verhalten“ wird hier im eigentlichen Sinne definiert. Hingegen wird in der Informatik „Verhalten“ oft im Sinne von „Handlungen“, also als beabsichtigt verstanden.

Interaktionen[1]. Sind diese Interaktionen zudem noch symbolisch (verbal, schriftlich usw.) so spricht man von *Kommunikation*.

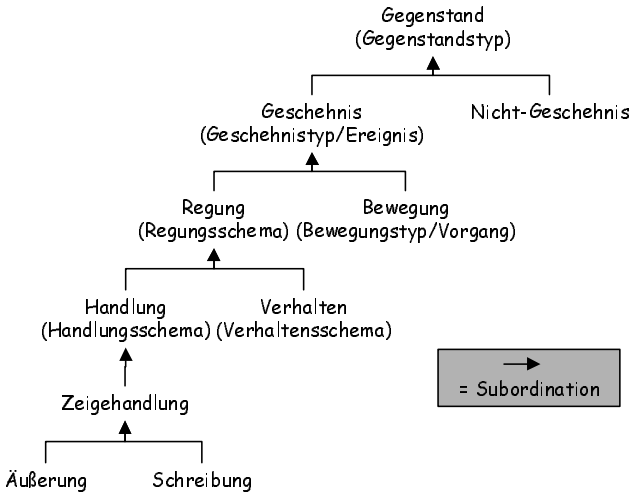


Abbildung 5.1: Gegenstandseinteilung

Soll nicht speziell von individuellen Gegenständen gesprochen werden, so kann auf einer abstrakten Ebene über *Typen* oder *Schemata* dieser Gegenstände ganz allgemein geredet werden. Konkrete Dinge und Geschehnisse werden dann auch als *Aktualisierungen* entsprechender Schemata verstanden.

Eine besondere Rolle spielen im Zusammenhang dieser Arbeit Geschehnistypen – auch *Ereignisse* genannt –, da sie ja auch die „Geschehnisse" von Softwareprodukten betrachtet. Ereignisse können aus einfacheren Ereignissen zusammengesetzt werden. So ist der Besuch eines Fußballspiels ein Ereignis, das aus Teilereignissen zusammengesetzt ist. Dieses Ereignis ist dann im Verlauf betrachtet (mit seinen Teilereignissen) ein *Vorgang*. Analog setzt sich ein Geschäftsprozess aus einer Reihe von Interaktionen (inklusive Kommunikationen) zusammen.

Geschehnisse lassen sich auch als Veränderungen von Zuständen beschreiben. Zustände sind Augenblicke zwischen verschiedenen Ereignissen, in denen vom Geschehen abstrahiert wird (vgl. Abschnitt 4.1.7).

Weitere Gegenstandseinteilungen werden in [Hartmann 1990] vorgenommen. Betrachtet man mehrere Geschehnisse – beispielsweise mehrere Handlungen, die in einem Zusammenhang stehen –, so kann man oft bezogen auf die einzelnen Geschehnisse resultierende

---

[1] Der Begriff „Interaktion" hat in der Informatik eine metaphorische Bedeutung. Softwarekomponenten *kommunizieren* miteinander und „handeln" folglich immer symbolisch. Ein Mischfall ist die Mensch-Computer-Interaktion.

Zustandskombinationen erkennen, die von Bedeutung sind. Solche Zustandskombinationen werden auch als *Situationen* bezeichnet. Situationsschemata beschreiben also Typen von Situationen. *Handlungsweisen* liegen dann vor, wenn ein Handlungsschema regelmäßig beim Vorliegen der Aktualisierung eines bestimmten Situationsschemas aktualisiert wird.

Eine *poietische Handlung* (Herstellungshandlung) ist eine Handlung, nach deren geglückter Ausführung „Spuren“ (*Artefakte*) zurückbleiben, nämlich die hergestellten Dinge.

Aufforderungsschemata (bzw. etwas ungenauer: *Aufforderungen*) sind Handlungsschemata, die bei ihrer Aktualisierung die Aktualisierung eines weiteren Handlungsschemas bewirken sollen. Das zweite Handlungsschema soll also befolgt werden. Aufforderungen wie auch das Grüßen, das Warnen und Drohen sowie das Mitteilen werden als Zeigehandlungsschemata verstanden. Zeigehandlungsschemata können u.a. durch Laut- und Schreibschemata oder auch „Malschemata“ vermittelt werden und heißen dann Sprachhandlungsschemata i.e.S. (oder auch „Diagrammhandlungsschemata“). Die einzelnen Aktualisierungen nennt man Äußerungen bzw. Schreibungen. *Schreibungen* gehören zu den poietischen Handlungsschemata. Die Spuren (Produkte) der Aktualisierungen der Schreibschemata heißen Marken. Wenn für alle Aktualisierungen eines Schreibschemas die gleichen Aussagen gelten, so spricht man auch von einem *Zeichen*. Komplexe Zeichen, sogenannte *Ausdrücke*, werden durch mehrere *Trägerhandlungsschemata* vermittelt. Diejenigen Ausdrücke, die für Sprachhandlungsschemata stehen, heißen Sätze.

5.4 Wortarten

Ohne auf die Herleitung der hier vorgestellten Normsprache (siehe dazu z.B. [Lorenzen 1987]) näher einzugehen, werden nun ihre wesentlichen Wort- und Satzarten vorgestellt. Diese Arten orientieren sich an der Gegenstandseinteilung und richten sich zudem an den germanischen Sprachen aus. Während die Gegenstandseinteilung kulturinvariant ist, werden die Wort- und Satzarten an bekannten Satzmustern ausgerichtet. Dadurch wird das Erlernen dieser Sprache wesentlich vereinfacht.

Wörter können grob in Nominatoren, Prädikatoren und Partikel unterteilt werden. *Nominatoren* benennen Gegenstände (Dinge oder Geschehnisse) eindeutig. Zu den Nominatoren zählen Eigennamen („Otto“, „Ilse“, „OL-PD 208“ als Bezeichner eines Autos), Leernamen („ich“, „du“) und durch Demonstratoren („dies“) gebildete Kennzeichnungen („dieses Auto“).

Prädikatoren sind Ausdrücke für Gegenstandsschemata. So kann man entsprechend der Gegenstandseinteilung Ding- und Geschehnisprädikatoren usw. unterscheiden. Diese Gegenstandsprädikatoren werden auch als *Eigenprädikatoren* bezeichnet. Ein Eigenprädikator ist ein Ausdruck, der sich Gegenständen z.B. durch Zeigehandlungen („Das ist ein Haus.“) oder darauf aufbauende Sprachregelungen wie Definitionen zuordnen lässt.

Zur genaueren Unterscheidung von Eigenprädikatoren werden *Apprädikatoren* genutzt, die Eigenprädikatoren vorangestellt werden. Entsprechend der Eigenprädikatoren können dann *Geschehnisapprädikatoren* (Adverbien) und *Dingapprädikatoren* (Adjektive) unterschieden werden.

Partikel sind entweder konstante grammatische Partikel, die als Hilfsverben der syntaktischen Unterscheidung der Satzarten (siehe Abschnitt 5.5) dienen, oder logische Partikel, die den Aufbau komplexer Sätze aus Elementarsätzen (siehe Abschnitt 5.7) ermöglichen.

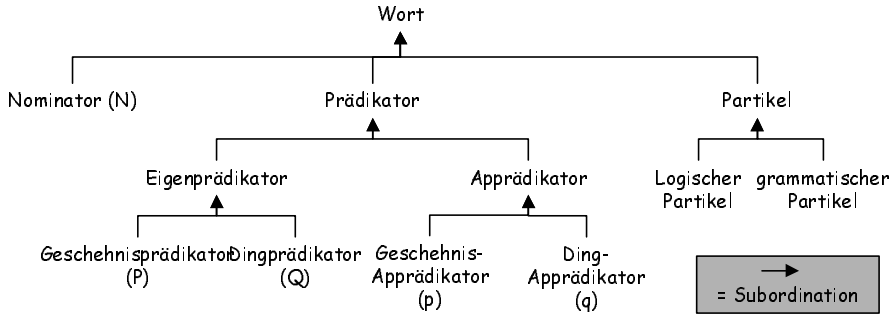


Abbildung 5.2: Wortarten

5.5 Elementarsätze

Im Folgenden werden die elementaren Satzarten von Normsprachen vorgestellt. Da Interaktionen zwischen Komponenten auf der Kommunikation von Sätzen basieren, stellen diese Satzarten für typische Interpretationen formaler Operationen bzw. Operationsaufrufe dar, wie sie später bei der Interpretation einer Komponente (siehe Kapitel 8) eingesetzt werden können.

Die Satzarten von Normsprachen (siehe Abbildung 5.3) können gemäß [Hartmann 1990]) in *Imperativ*- (Aufforderungen) und *Indikativsätze* (Aussagen) unterteilt werden. Imperativsätze können final und afinal sein. In finalen Imperativsätzen ist in der Aufforderung der Zweck enthalten. Der Auffordernde fordert also die Herbeiführung von Sachverhalten. In afinalen Imperativsätzen wird nur die auszuführende Handlung aufgeführt. Indikativsätze können entsprechend der Gegenstandseinteilung in *Geschehnisaussagen* und *Seinsaussagen* (auch *Zustandsaussagen* genannt) unterteilt werden. Geschehnisaussagen sind entweder *Tat*- oder *Bewegungsaussagen*. Tataussagen (auch Regungsaussagen) betonen die aktive Beteiligung eines Gegenstandes an der Tat, während Bewegungsaussagen eine Passivität des bewegten Gegenstandes unterstellen. Tataussagen kann man weiter in Handlungsaussagen und Verhaltensaussagen unterteilen. Handlungen werden bewusst durchgeführt, während das Verhalten – beispielsweise einem Tier – einfach widerfährt.

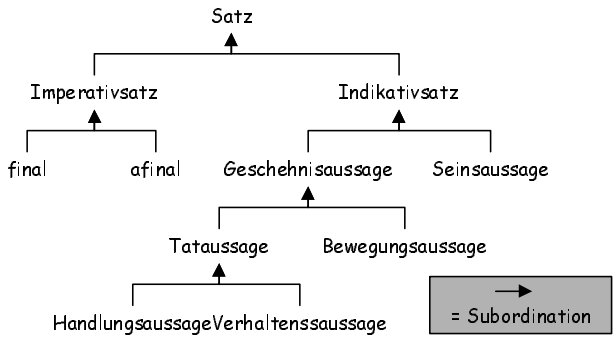


Abbildung 5.3: Satzarten

Methodisch stehen Imperativsätze, also Aufforderungen, mit denen der Sender beim Empfänger etwas erreichen will, am Anfang. Der Sender fordert beispielsweise durch die Äußerung eines Geschehnisprädikators eine Handlung des Empfängers. Um später Aufforderungen von bloßen Aussagen unterscheiden zu können, wird das Ausrufungszeichen („!“) als Appellator als erstes Satzzeichen eingeführt.

Die Aufforderung

```
! laufen
```

soll beispielsweise dem Empfänger anzeigen, dass er laufen soll.

Neben dem Appellator für Imperativsätze gibt es, wie in Abschnitt 5.4 beschrieben, grammatische Partikel, sogenannte *Kopulae*, zur Unterscheidung verschiedener Arten von Aussagesätzen. Folgende Partikel (siehe auch Tabelle 5.1) können mindestens unterschieden werden:

- Seinskopula ε (lies: ‚ist‘),
- Geschehniskopula κ (lies: ‚ist am‘),
- Tatkopula π (lies: ‚tut‘),
- Fähigkeitskopula γ (lies: ‚kann‘),
- Widerfahrniskopula τ (lies: ‚wird‘) und
- Teilungskopula σ (lies: ‚hat‘).

Aus diesen Kopulae lassen sich nun in Verbindung mit den zuvor dargestellten Wortarten Indikativsätze aufbauen:

```
[Peter] π [werfen]      (lies: Peter tut werfen)
[Peter] τ [werfen]      (lies: Peter wird geworfen)
```

Komplexere Imperativsätze können dann durch Voranstellen eines Appellators (‚!‘) vor einen Aussagesatz gebildet werden:

```
! [Peter] π [werfen]
```
(lies: Mach, dass Peter tut werfen)

Satzart	Partikel	Bemerkung
Imperativsatz	!	Durch Voranstellen von ‚!‘ wird ein Imperativsatz gekennzeichnet. (Es wird hier vernachlässigt, dass Indikativsätze aus Imperativsätzen konstruiert werden.)
Tataussagen	π	Die Unterscheidung, ob eine Tataussage eine Handlung oder ein Verhalten beschreibt, erfolgt aus dem Kontext: Handlungen entstehen durch eine Aufforderung.
Bewegungsaussagen	κ	Da man Bewegungsaussagen aus Tataussagen konstruieren kann, kann κ (Geschehniskopula) allgemein für Geschehnisaussagen stehen.
Seinsaussagen	ε, σ, γ	‚ε‘ weist auf (intensionaler Ebene) einem Gegenstand einen Prädikator zu. ‚σ‘ definiert eine Teile-Ganzes-Beziehung, und ‚γ‘ spricht einem Prädikator eine Fähigkeit zu.

Tabelle 5.1: Satzarten und Kopulae

Bislang wurden relativ einfache Sätze betrachtet, die durch das Satzmuster

```
[N] K [P]      (Nominator, Kopula und Prädikator)
```

dargestellt werden können. Etwas mächtigere Elementarsätze können durch die Verwendung von Apprädikatoren und das Hinzufügen von Objekten (z.B. Dingprädikatoren) zu Tatprädikatoren aufgebaut werden:

```
[Peter] π [weit werfen] [weiß Stein]
```
(lies: Peter tut weit werfen weißen Stein)

„Weit“ ist hier ein Geschehnisapprädikator und „weiß“ ein Dingapprädikator. „Stein“ ist das direkte Objekt zum Tatprädikator „werfen“. Die Satzmuster lassen sich also um Apprädikatoren und direkte Objekte wie folgt erweitern:

```
[N] K [ p P] [q Q].
```

Weitere (indirekte) Objekte können durch sogenannte Kasusmorpheme oder durch Präpositionen angehängt werden. Man kann mindestens drei verschiedene Fälle von Kasusmorphemen unterscheiden:

I) Mittelfall: Womit wird etwas gemacht?

II) Werkfall: Zu was wird etwas gemacht/verarbeitet?

III) Gebefall: An wen oder was wird etwas gegeben?

Beispiel:

```
[Peter] π [sprengen] [Garten] [mit- Gartenschlauch]
```

Mit Präpositionen lassen sich lokale Aussagen (wie z.B. mit „unten“, „oben“, „von“, „zu“) oder temporale Aussagen („jetzt“, „früher“, „später“) machen.

5.6 Bezug zur formalen Logik

Bislang wurden elementare Sätze mit Subjekt, Kopula, Prädikator und bis zu drei Objekten eingeführt. Die endgültige Satzform hat dann folgendes Aussehen:

```
[S] K [P] [O1] [O2] [O3]
```

Für den Logiker ist die Differenzierung von Taten, Bewegungen und Dingen meist irrelevant. Er abstrahiert deshalb von der Satzart, die durch die Kopula angezeigt wird, und führt dazu alle Aussagen auf bloße Seinsaussagen zurück. So gelangt man zu folgender standardisierter Satzform:

```
S1 S2 S3 S4 ε P   oder auch: P(S1,S2,S3,S4)
```

Eine Aussage wird so als 4- (bzw. allgemein als n-)stelliges Prädikat dargestellt. P wird n-stelliger Prädikator oder auch *Relator* genannt. Diese Schreibweise entspricht dann der von mehrstelligen Prädikaten in der Prädikatenlogik.

Es ist unmittelbar nachvollziehbar, dass durch diese Abstraktion Informationen – nämlich, ob es sich bei einer Aussage um eine Seins- oder eine Tataussage handelt und in welcher Beziehung die Objekte zueinander stehen – verloren gehen.

Mit dem gleichen Problem hat man auch bei der Abstraktion im Rahmen der Softwareentwicklung und bei der Wiederverwendung zu tun. So können z.B. Methodenaufrufe zum Mitteilen von Aussagen oder aber zur Aufforderung an andere Objekte dienen. Um was für eine Satzart es sich jedoch konkret handelt (vgl. Abschnitt 5.1), ist i.d.R. nicht mehr zu erkennen.

Interessant ist in diesem Zusammenhang auch die Beobachtung von Hartmann ([Hartmann 1990]), dass es selten elementare Relatoren gibt, die mehr als vier Stellen besitzen. Diese Beobachtung lässt sich auch auf die Hornklausel-Definitionen von PROLOG-Programmen (siehe [Clocksin und Mellish 1987]) übertragen.

5.7 Komplexe Satzarten von Normsprachen

Komplexe Sätze werden durch die Verwendung logischer Partikel aus Elementarsätzen aufgebaut. Dazu zählen der *Negator* (¬, „nicht _“), der *Konjunktor* (∧, „_ und _“), der *Adjunktor* (∨, „_ oder _“), der *Subjunktor* ($\rightarrow_C$, „wenn _, dann _“, siehe auch Abschnitt 5.10). Weitere logische Partikel lassen sich ableiten. So gibt es z.B. noch den Existenzquantor (∃, „es gilt für mindestens einen, _“) und den Allquantor (∀, „es gilt für alle, _“), die abkürzende Schreibweisen für große Adjunktionen und Konjunktionen darstellen, sowie den *Disjunktor* (⊗, „entweder _ oder _“). Auf die Bedeutung wird hier nicht genauer eingegangen. Es sei nur gesagt, dass hier einer klassischen, aussagenlogischen Interpretation gefolgt wird.

5.8 Grammatik

Im Folgenden soll die rationale Grammatik der Normsprache detailliert aufgeführt werden. Eine rationale Grammatik definiert eine *Tiefenstruktur* – vergleichbar einem Ableitungsbaum einer Grammatik (siehe Beispiel in Abbildung 5.4) – zu einem Satz S. Diese Tiefenstruktur[1] ist eindeutig, d.h. zu einem Satz gibt es immer nur einen Ableitungsbaum. Die natürlichsprachliche Form des gleichen Satzes kann dann als Oberflächenstruktur bezeichnet werden ([Hartmann 1990]). Sie ist hingegen in der Grammatik der deutschen Sprache nicht eindeutig.

```
(1)  Satz ::= Aufforderung | Aussage

(2)  Aufforderung  ::= "!" Aussage

(3)  Aussage   ::= Geschehnisaussage | Zustandsaussage

(4)  Geschehnisaussage ::= Tataussage | Bewegungsaussage

(5)  Tataussage    ::= Handlungsaussage | Verhaltensaussage

(6)  Handlungsaussage  ::= Subjekt "πH" GPrädikation Obj1 Obj2 Obj3

(7)  Verhaltensaussage ::= Subjekt "πV" GPrädikation Obj1 Obj2 Obj3
```

[1] In den normsprachlichen Satzbeispielen, die im Weiteren angeführt werden, wird auf die eckigen Klammern („[“, „]“), die normalerweise Bestandteil der Syntax sind und die Tiefenstruktur betonen, verzichtet, wenn der Satz auch so eindeutig zu verstehen ist.

```
(8)  Bewegungsaussage  ::= Subjekt “κ” GPrädikation Obj1 Obj2 Obj3

(9)  Zustandsaussage   ::= Subjekt “ε” Prädikation

     Subjekt        ::= “[” Nominator “]”
     Nominator      ::= Eigenname | Leername | Demonstrator
     Eigenname      ::= “Erwin” | “Ilse” | “OL-PD 208” | ...
     Leername       ::= “ich” | “du” | ...
     Demonstrator   ::= “dies” Prädikator

     GPrädikation   ::= “[” GPrädikationLoop “]”
     GPrädikationLoop  ::= GPrädikator
                    |   GAPrädikator[“+”|“-”|“#”] GPrädikationLoop

     Obj1           ::= “[]” | “[” ( Nominator | Obj1Loop ) “]”
     Obj1Loop       ::= DPrädikator
                    | DAPrädikator[“+”|“-”|“#”]“,” Obj1Loop

     Obj2           ::= “[]” | “[” Obj2Loop “]”
     Obj2Loop       ::= Obj2Part | Obj2Part“,” Obj2Loop
     Obj2Part       ::= KMorphem ( Nominator | Obj1Loop)
     KMorphem       ::= “mit-” | “zu-” | “als-” | “cum-” | “durch-”
                                  /*“durch-” nur im Passiv*/
     Obj3           ::= “[]” | “[” Obj3Loop “]”
     Obj3Loop       ::= Obj3Part | Obj3Part“,” Obj3Loop
     Obj3Part       ::= Lokalpräpositionen Nominator
                    | Tempppräposition Nominator
     Lokalpräpositionen ::= Lokalpräposition
                    | Lokalpräposition Lokalpräpositionen
     Lokalpräposition  ::= “auf” | “unter” | “innen” | “außen”
                    | “von” | “zu” |
     Tempppräposition   ::= “BIS” | “AB” | “VOR” | “NACH”

     GPrädikator    ::= Prädikator
     DPrädikator    ::= Prädikator [Besitzanzeiger] [Teilanzeiger]

     GAPrädikator   ::= Apprädikator
     DAPrädikator   ::= Apprädikator

     Prädikation    ::= “[” PrädikationsLoop “]”
     PrädikationsLoop  ::= Prädikator | Apprädikator“,” PrädikationsLoop

     Prädikator     ::= “o” | [“A”-“Z]+

     Besitzanzeiger::= “~” Nominator
     Teilanzeiger   ::= “⊂” (Nominator | Prädikation)
```

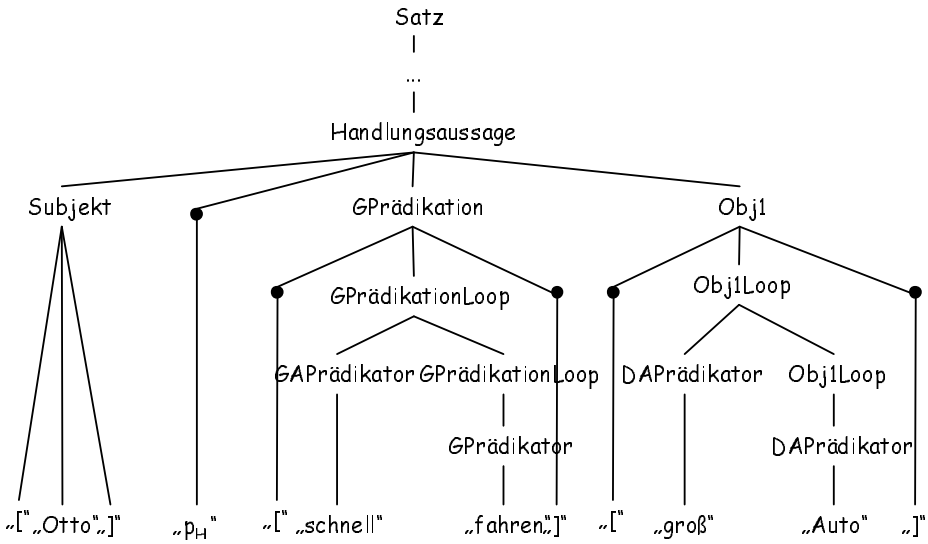


Abbildung 5.4: Beispiel eines Ableitungsbaumes (Tiefenstruktur)

5.9 Termini, Terminologie, Definition

Durch Prädikatorenregeln kann die Verwendung von Prädikatoren normiert werden. Diese Regeln basieren nicht auf Erfahrungen – sie können also nicht empraktisch erlernt werden –, sondern auf Sprachregelungen einer Gruppe von Menschen (Sprachgemeinschaft). Ein Prädikator, dessen Verwendung im Zusammenhang mit anderen Prädikatoren über solche Regeln geregelt wird, heißt *Terminus*. Eine *Terminologie* besteht aus den Prädikatoren inklusive der Verwendungsregeln.

Im Zusammenhang dieser Arbeit können so Abstraktionen von Begriffen formuliert werden. Ein wesentliche Erleichterung erreicht man dadurch, dass man in den Regeln Variablen benutzt.

Der Ausdruck „Eine Rose ist eine Blume.“ ist ein Beispiel einer umgangssprachlichen Prädikatorenregel und nicht etwa eine Elementaraussage. Normsprachlich hat die Regel – in der Welt der Programmierung würde man von einer „is-a“-Beziehung sprechen – folgende Form:

$$x \in \mathrm{Rose} \Rightarrow x \in \mathrm{Blume}$$

Allgemein:

$$x_1, \ldots, x_n \in P_1 \Rightarrow x_1, \ldots, x_n \in P_2$$

Definitionen sind spezielle Prädikatorenregeln, die der Abkürzung langer Ausdrücke dienen.

$$x_{1,1}, \ldots, x_{n1} \in P_1 \ ;$$

$x_{2,1}$, ..., x_{n2} ε P_2 ; $\Leftrightarrow$ x_1, ..., x_n ε P
...
$x_{m,1}$, ..., x_{nm} ε P_m

Mit Definitionen lassen sich Lexika aufbauen, die später bei der Interpretation eines komponentenbasierten Anwendungssystems genutzt werden können.

5.10 Beschreibung kausaler Zusammenhänge

Beschreibungssprachen für Geschehnisse i. Allg. bzw. Handlungen und Verhalten im Speziellen stellen verschiedene Kausal-Operatoren wie Sequentialisierung und Parallelisierung von Geschehnissen zur Verfügung. Hierzu stellt sich die Frage, wie sich diese Operatoren konstruktiv herleiten lassen? Lorenzen (siehe [Lorenzen 1987]) weist darauf hin, dass die klassische Logik nicht ausreicht, um über Zukünftiges zu reden. Zukünftiges ist jedoch wesentlicher Bestandteil, um kausale Zusammenhänge zu beschreiben. Wenn jemand sagt: „Ich fahre nach Oldenburg.“, so kann dieser Satz als in die Zukunft gerichtetes Versprechen verstanden werden. Ein *Versprechen* ist somit eine Aufforderung einer Person an sich selbst, etwas zu befolgen:

Peter ! Peter π fahren zu-Oldenburg.

Da der Adressat diese Aufforderung oft nicht sofort befolgen kann, verspricht er dafür deren Befolgung. Als abkürzendes „Versprechungszeichen“ führt Lorenzen den (effektiven) *Subjunktor* „$\rightarrow$“ ein. Der effektive Subjunktor unterscheidet sich von dem Subjunktor der klassischen Logik „$\rightarrow_C$“ durch seine strikte Sequentialität. Es gilt $A \rightarrow B \neq \neg A \vee B$ (= $A \rightarrow_C B$), wodurch sich die beiden Subjunktoren klar voneinander unterscheiden. Das oben angeführte Versprechen würde wie folgt aussehen:

$\rightarrow$ Peter π fahren zu-Oldenburg.

oder allgemeiner:

$\rightarrow$ Subjekt “π_H” GPrädikation Obj... .

Da hier nur der Sonderfall, nämlich das unbedingte Versprechen, eingeführt wurde, leuchtet unmittelbar ein, warum das Pfeilsymbol gewählt wurde. Bei bedingten Versprechen kann man die Bedingung, die der Versprechende einhalten soll, vor den Subjunktor stellen. Ein Versprechen, das ein Subjekt 2 gegenüber einem Subjekt 1 macht, hat dann folgende Form haben:

Subjekt$_1$ “π_H” GPrädikation$_1$ Obj... $\rightarrow$ Subjekt$_2$ “π_H” GPrädikation$_2$ Obj...

Subjekt 1 und Subjekt 2 können natürlich auch identisch sein, wodurch die Handlung eines Subjektes bedingt versprochen wird: „Wenn ich das Buch gelesen habe, schicke ich es Dir“.

Die Grammatik kann jetzt um diese komplexen Aussagen für sequentiell auszuführende Handlungen erweitert werden:

```
Sequenz   ::=  "→" KAussage
          |        KAussage "→" Sequenz
```

KAussagen sind einfache Aussagen (Aussage), wie sie schon vorgestellt wurden, oder komplexe Aussagen, die aus sequentiell, parallel oder disjunktiv verknüpfte einfacheren Aussagen bestehen. KAussage ist also rekursiv definiert:

```
KAussage ::=  Aussage | Sequenz | Parallel | Disjunktion
```

Versprechungen (oder abstrakter: Sequenzen von Handlungen) lassen sich nun mittels des Subjunktors und der zuvor eingeführten elementaren logischen Partikel zu weiteren Junktoren zusammensetzen. Parallelität, wie sie auch zur Beschreibung formaler Prozesssysteme (siehe Kapitel 17) genutzt wird, kann durch eine beliebige Sequenz zweier Geschehnisse formuliert werden.

$A \,||\, B := A \rightarrow B \wedge B \rightarrow A$ (Bisubjunktor)

Wichtig ist bei der Definition paralleler Geschehnisse der Aspekt der Synchronisation. Synchronität kommt dadurch zustande, dass zwei Subjekte gemeinsam handeln. Diese Forderung wird durch das "Matchen" zweier Handlungsschemata umgesetzt (siehe Abschnitte 17.1.2 und 17.2.3). In abstrakteren Beschreibungssprachen, in denen über die elementaren Aktionen keine Aussagen getroffen werden können, müssen Synchronisationspunkte explizit festgelegt werden.

Alternative Ausführungen können „klassisch" logisch als exklusives Oder verstanden werden:

$A \sqcup B := A \vee B \wedge \neg(A \wedge B)$ (Disjunktor)

Die Grammatik wird um folgende Regeln erweitert:

```
Parallel      ::= KAussage
                  |      KAussage "||" KAussage
Disjunktion   ::= KAussage
                  |      KAussage "xor" KAussage
```

KAussagen sind komplexe oder einfache Aussagen:

```
KAussage ::=  Aussage | Sequenz | Parallel | Disjunktion
```

5.11 Fazit

In diesem Kapitel wurde eine Normsprache vorgestellt, die der natürlichen Sprache sehr ähnlich ist, jedoch um „Defekte" wie z.B. Mehrdeutigkeit bereinigt wurde. Normsprachliche

Sätze lassen sich einfach in natürlichsprachliche Sätze überführen, indem die Flexionen (Konjugation und Deklination) der Prädikatoren berücksichtigt werden.

Idee dieser Arbeit ist es, eine Normsprache zur Angabe einer Interpretation eines formalen Systems, nämlich der Beschreibung eines komponentenbasierten Anwendungssystems, zu nutzen. Dieser Ansatz wird in Kapitel 8 vorgestellt, nachdem zuvor als Fundament eine entsprechende formale Sprache (Kapitel 6 und 7) zur Beschreibung von Komponenten hergeleitet und entwickelt wurde.

Teil II Prozessorientierte Konfiguration von Anwendungssystemen

6 Konkretisierung der Problemstellung

In der Einleitung wurden in Abschnitt 1.2 die Rollen *Komponentenhersteller*, *Anwendungsarchitekten*, *Konfigurierer* und *Anwender* der komponentenbasierten Softwareentwicklung definiert. Das Hauptaugenmerk dieser Arbeit liegt auf der Kommunikation zwischen Anwendungsarchitekten und Konfigurierern sowie Konfigurierern und Anwendern bzw. den dazu benötigten Hilfsmitteln. Es soll im Folgenden (siehe Abschnitte 6.1 – 6.3)genauer darauf eingegangen werden, welche Anforderungen von Seiten dieser drei Rollen gestellt werden bzw. welche Anforderungen Anwendungsarchitekten und Konfigurierer für die jeweils nächste Rolle, also Konfigurierer bzw. Anwender, erfüllen müssen. In einem weiteren Abschnitt (6.4) werden die technischen Anforderungen aufgeführt, die aus Sicht der sich heute abzeichnenden Komponententechnologie relevant sind.

Aus diesen Anforderungen werden dann in Kapitel 7 die Entwurfsentscheidungen für die Sprachen zur Beschreibung von Komponenten und Anwendungssoftwareprodukten abgeleitet.

6.1 Anforderungen des Anwendungsarchitekten

Anwendungsarchitekten erstellen konfigurierbare Anwendungssoftwareprodukte aus kleineren Komponenten, indem sie diese zu komplexeren Komponenten zusammensetzen. Dabei wird nicht mit den eigentlichen Produkte, sondern mit abstrakteren Beschreibungen der Komponenten gearbeitet. Die Beschreibungen abstrahieren im Sinne einer Black-Box-Abstraktion (siehe Abschnitt 3.2.7) von den für den Kompositionsprozess unwichtigen Details der Komponenten und erlauben so einen schnelleren Überblick über deren entscheidenden Eigenschaften.

Eine komplexe Beschreibung stellt ein aus den zugehörigen Komponenten konstruiertes Anwendungssoftwareprodukt dar. Wesentliches Merkmal einer Sprache zur Beschreibung von Komponenten ist also ihre *Kompositionalität*. Sie fordert, dass die resultierende Beschreibung auch wirklich eine gültige Beschreibung des zusammengesetzten Produkts ist. Können Komponenten nicht ohne zusätzliche Programmierung (z.B. Scripting, siehe 3.2.8) miteinander kombiniert werden, so muss der Anwendungsarchitekt – nun in der Rolle eines Komponentenherstellers – eine geeignete Komponente für diese Aufgabe entwickeln.

Eine Beschreibungssprache für Komponenten sollte nicht nur ein spezielles *Komponentenmodell* unterstützen, sondern möglichst weit und gut von möglichen Technologien und Programmiersprachen abstrahieren. So sollte es z.B. sowohl heute vorherrschende imperative als auch objektorientierte Modelle (siehe Abschnitt 3.3.1) unterstützen.

Die erstellten Anwendungssoftwareprodukte – und damit ihre Beschreibungen – sind im Gegensatz zu den meisten ihrer konstituierenden Komponenten(beschreibungen) domänenabhängig. D.h., während die kleineren Komponenten noch in mehreren Bereichen (Domänen) eingesetzt werden können, wird der Einsatz durch die Integration in ein Anwendungssoftwareprodukt auf den Anwendungsbereich dieses Produktes zunehmend eingeschränkt. Aus diesem Grund sollte eine Sprache zur Beschreibung von Komponenten bzw. der daraus aufgebauten Anwendungssoftwareprodukte eine domänenspezifische *Spezialisierung* (Interpretation) der Beschreibungen erlauben.

Monolithische Anwendungssoftwareprodukte (siehe Abschnitt 2.1.3) sind nicht sofort einsatzbereit. Vielmehr müssen sie an das individuelle Unternehmen angepasst werden, da selbst vermeintlich ähnliche Unternehmen einer Branche sich in Details immer wieder wesentlich voneinander unterscheiden und dies als Wettbewerber auch unverändert so wollen. Will man den Anwendern den vollständigen Aufbau eines Systems aus Komponenten ersparen, so ist es sinnvoll, dass auf dem zukünftigen Softwaremarkt vorgefertigte, komponentenbasierte Anwendungssoftwareprodukte angeboten werden, die Möglichkeiten zur Erweiterung und zur Auswahl alternativer Implementierungen bieten werden. Deshalb sollte eine Sprache zur Beschreibung von Anwendungssoftwareprodukten Möglichkeiten zur Beschreibung von *Konfigurationsalternativen* bieten.

6.2 Anforderungen des Konfigurierers

Der Konfigurierer ist ein Experte für einen bestimmten Anwendungsbereich, z.B. eine Branche. Seine Aufgabe ist es, ein Anwendungssoftwareprodukt, das aufgrund seiner Eigenschaften grundsätzlich für einen Anwendungsbereich geeignet ist – dieser Nachweis muss zuvor im Rahmen eines Softwareauswahlprozesses erbracht werden – auf die Bedürfnisse eines konkreten Einsatzbereiches anzupassen. Dafür ist es hilfreich, wenn er

- verschiedene Konfigurationen interaktiv „durchspielen“ kann,
- die zukünftig mit dem Softwareprodukt abzuwickelnden Geschäftsprozesse schnell und einfach beschreiben kann sowie
- die Terminologie des anwendenden Unternehmens in die Beschreibungen des Anwendungssoftwareprodukts einbringen kann, da diese trotz der Zugehörigkeit eines Unternehmens zu einer Branche immer wieder von Unternehmen zu Unternehmen differieren kann.

Der erste und der dritte Punkt entsprechen den Anforderungen des Anwendungsarchitekten (siehe Abschnitt 6.1), die Beschreibungen von Anwendungssoftwareprodukten domänenspezifisch spezialisieren zu können. Die zweite Anforderung zielt auf die in Unternehmen zunehmend anzutreffende geschäftsprozessorientierte Ausrichtung der Anwendungssysteme. Denn in der effizienten und effektiven Gestaltung der

Geschäftsprozesse liegen heute meist die Hauptpotentiale neuer Anwendungssoftwareprodukte (vgl. Hypothese 4 in Abschnitt 1.4). Aus diesem Grund sollten die Beschreibungen von Geschäftsprozessen – zumindest soweit sie durch das Anwendungssoftwareprodukt unterstützt werden – möglichst einfach abgeleitet werden können.

6.3 Anforderungen des Anwenders/Benutzers

Der Anwender bzw. das anwendende Unternehmen benötigt eine an den Geschäftsprozessen ausgerichtete Dokumentation seiner Anwendungssysteme, denn Unternehmen richten ihre Organisation zunehmend an diesen Prozessen statt an den funktional orientierten Abteilungen aus. Anhand dieser Geschäftsprozessbeschreibungen können – zumindest teilweise – die Konsequenzen einer neuen Software für ein Unternehmen veranschaulicht und bewertet werden. Workflowmanagement und Prozesskostenrechnung sind nur zwei der Aufgabengebiete eines Unternehmens, die sich außerdem diese Beschreibungen zunutze machen können.

Natürlich hat auch der einzelne Benutzer durch eine prozessorientierte Dokumentation der Unternehmensorganisation einen Nutzen. So sollten sich aus den Produktbeschreibungen relativ einfach Ablaufbeschreibungen – z.B. für Organisationsanweisungen – für die verschiedenen Aufgaben der Mitarbeiter ableiten lassen, die sich zum einen zur Bedienung der Softwareprodukte und zum anderen zur Einordnung der Tätigkeit in den Gesamtkontext des Unternehmens verwenden lassen.

Für den Benutzer hat deswegen die *Verständlichkeit* der erzeugten Beschreibungen einen besonders hohen Stellenwert.

6.4 Voraussetzungen der Komponententechnologie

Komponenten sind in dieser Arbeit die elementaren Gegenstände der Betrachtung und Beschreibung. Deshalb sollen einige, nachfolgend konkretisierte Erkenntnisse, die sich im – allerdings noch jungen – Forschungsumfeld komponentenbasierter Softwareentwicklung herauskristallisiert und als praktikabel erwiesen haben, als gegeben akzeptiert werden.

Ein Aspekt ist dabei die Diskussion um das Verhältnis zwischen Komponenten- und Objekttechnologie. Dabei kann verkürzt (siehe ansonsten Abschnitt 3.2.4) folgende Anforderung festgehalten werden: Eine Komponentenmodell sollte nicht zwingend ein Objektmodell voraussetzen. Wenn Objekte bzw. Klassen als kleinere Elemente für eine Komponente von Bedeutung sind, so werden sie als weitere Schnittstellen zur Funktionalität einer Komponente gesehen. D.h., Klassen bzw. Objekte gehören immer genau zu einer Komponente und werden nur als Teile dieser Komponenten betrachtet. Jedoch können

Komponenten auch ausschließlich auf klassischen, imperativen Sprachparadigmen basierend beschrieben werden.

Komponenten haben keinen persistenten Zustand. Eine Komponente kann also nicht während ihrer Laufzeit in einem geänderten Zustand abgespeichert werden. Allerdings können selbstverständlich Objekte oder Daten, die von einer Komponente erzeugt wurden, persistent gespeichert werden. *Konfigurationsinformationen*, also beispielsweise Informationen darüber, welche Komponenten mit welchen anderen Komponenten parametriert wurde, gehören nicht zu den Komponenten selbst und werden deswegen separat verwaltet.

Komponenten werden immer komplett in Struktur und Verhalten beschrieben. Die Beschreibungen werden dabei als *Black-Box-Abstraktionen* der Komponenten betrachtet. Die Beschreibung der Komponente sagt also nichts über ihre Implementierung aus, sondern beschreibt ausschließlich die nach außen hin sichtbaren Aufrufschnittstellen sowie das Verhalten. Die Verhaltensbeschreibung definiert die gültigen Reihenfolgen von Schnittstellenaufrufen zwischen den Komponenten. Um grundsätzlich jedes Verhalten einer Komponente beschreiben zu können, wird eine turing-mächtige Sprache benötigt. Sie muss beispielsweise Bedingungen sowie Schleifenoperatoren (oder Rekursion) anbieten.

Es sollte klar zwischen den Konzepten „Komponente“ und „Kontrakt“ unterschieden werden. Eine Komponente ist ein konkretes Element der Softwareentwicklung, die in eine andere Komponente eingesetzt werden kann. Kontrakte spezifizieren die Eigenschaften, die Komponenten haben müssen, damit sie an entsprechenden Stellen eingesetzt werden können. Ein Kontrakt definiert, wann eine Komponente „kompatibel“ ist. Eine Komponente, für die es keinen Kontrakt gibt, ist sinnlos und überflüssig, da es keine Stelle gibt, an der sie eingesetzt werden kann.

Die Prinzipien der *Implementierungs- und Schnittstellenvererbung* (siehe Abschnitt 3.1.2) werden für eine Beschreibungssprache für Komponenten nicht benötigt. Sie könnten zwar die Effizienz bei der Erstellung von Beschreibungen steigern, sagen aber grundsätzlich nichts über Kompatibilität und Substituierbarkeit aus. In dieser Arbeit wird jede Komponente vollständig, d.h. ohne sich auf die Beschreibung einer anderen Komponente zu stützen, beschrieben und kann so zu verschiedenen Kontrakten kompatibel sein, ohne diese als Subtypbeziehung explizit angeben zu müssen. Dadurch wird ermöglicht, dass Komponenten auch in Bereichen eingesetzt werden können, für die sie ursprünglich gar nicht vorgesehen waren – vorausgesetzt sie halten entsprechende Kontrakte ein.

Die schon in Abschnitt 4.1.9 geforderte Kompositionalität setzt voraus, dass Komponenten aus anderen Komponenten zusammengesetzt werden können. Zur Umsetzung dieser Anforderung gibt es grundsätzlich zwei Alternativen: Entweder einige Komponenten besitzen Steckplätze, in die andere Komponenten eingesetzt werden können oder es gibt spezielle Komponenten (Konnektoren) wie z.B. in Darwin (siehe Abschnitt 4.4.1), die „normale“

Komponenten miteinander verbinden. Offensichtlich ist der zweite Fall ein Spezialfall des ersten, da Konnektoren als Komponenten mit zwei oder mehr Steckplätzen verstanden werden können.

6.4.1 Schnittstellen

Schnittstellen lassen sich, wie in Abschnitt 3.3.1 beschrieben, in direkte und indirekte Schnittstellen unterteilen. Für eine Kombination aus beiden Schnittstellenarten, wie sie hier angestrebt wird, spricht, dass noch nicht abzusehen ist, welche derzeit diskutierten Komponentenmodelle sich durchsetzen werden. Industriestandards wie DCOM von Microsoft wie auch die Vorschläge der OAG gehen den Weg direkter Schnittstellen. Jedoch setzt sich zunehmend die objektorientierte Programmierung und damit u.U. auch ihre Komponentenmodelle wie die der OMG und Enterprise JavaBeans (EJB) von SUN durch, die eher indirekte Schnittstellen unterstützen. Es ist jedoch zu beachten, dass die objektorientierten Komponentenmodelle i.d.R. auf statische Methoden, die letztlich nichts anderes als direkte Schnittstellen sind, verzichten können. So besitzen beispielsweise die EJB-Komponenten (Enterprise Beans) Methoden zum Erzeugen, Finden und Löschen von Objekten, die unabhängig von einem bestimmten Objekt sind und somit direkt der Verwaltung von Objekten dienen.[1]

6.4.2 Zuordnung der Verhaltensbeschreibungen

Für die Beschreibungssprache für Komponenten bedeutet die Verfügbarkeit direkter und indirekter Schnittstellen, dass es möglich sein muss, das Verhalten sowohl aus direkter als auch indirekter Sicht zu beschreiben. Daraus lassen sich auch zwei Ebenen der Verhaltensbeschreibung einer Komponente ableiten: eine Ebene beschreibt das Verhalten der Komponente bezüglich der direkten Schnittstellen und die zweite Ebene beschreibt das Verhalten der durch die Komponente verwalteten Objekte. Auf die zweite Ebene kann verzichtet werden, wenn z.B. gar kein objektorientiertes Modell zugrunde liegt oder vom Verhalten der Objekte abstrahiert werden soll. Die beiden Ebenen der Verhaltensbeschreibung können einander referenzieren, so dass das Verhalten der Gesamtkomponente durch die Kombination der direkten und indirekten Verhaltensbeschreibungen beschrieben wird.

---

[1] Tatsächlich sind diese direkten Schnittstellen durch ihr Interface den Objekten zugeordnet, die durch eine Enterprise Bean verwaltet werden. Sie liefern allerdings immer die gleichen Ergebnisse, unabhängig davon, von welchem Objekt sie aufgerufen werden. Es handelt sich hier also um eine „Design-Nachlässigkeit“ bei der EJB-Spezifikation (siehe [Matena und Hapner 1999]). Besser wäre es gewesen, ein zentrales Objekt (im Sinne einer abstrakten Fabrik, siehe [Gamma et al. 1995]) mit diesen Schnittstellen auszustatten oder auf statische Methoden zurückzugreifen.

Eine Komponente besteht somit aus einer zentralen und mehreren lokalen Verhaltensbeschreibungen für die durch die Komponente bereitgestellten Objekte. Objekte gehören einer Klasse an, die einen Teilvertrag des Gesamtkontraktes darstellen. Diese Verträge legen die Signatur der Schnittstellen und das Verhalten, also die gültigen Kommunikationssequenzen der Schnittstellen eines Objektes dieser Klasse, fest.

Ein Vertrag ist somit nichts anderes als eine abstrakte Komponente, deren Verhaltensbeschreibung durch eine Komponente, die diesen Vertrag erfüllen soll, eingehalten werden muss. Der Vertrag beschreibt ausschließlich das aufrufbare Verhalten (*invocable behaviour*, siehe 4.1.2).

6.4.3 Zustand versus Geschehnis

Grundsätzlich können zwei Extreme der Verhaltensbeschreibung einer Komponente unterschieden werden. Bei der einen, der zustandsbezogenen (deklarativen) Beschreibungsform stehen die aus dem Verhalten resultierenden Zustände im Vordergrund, bei der anderen, der geschehnisbezogenen Beschreibungsform (z.B. in Form von Transitionen) dominieren die Handlungen der Akteure. Die zustandsbezogenen Beschreibungen gehen davon aus, dass Geschehnisse zu bloßen Ereignissen abstrahiert werden, deren innere Struktur nicht mehr von Interesse ist, die aber durch ihre eingrenzenden Zustandspaare ausreichend beschrieben werden. Die geschehnisbezogene Beschreibung legt ihr Gewicht auf die Beschreibung der Geschehnisse, die von einem Zustand in den anderen führt. Die Zustände sind hier abstrakte Gegenstände, deren Bedeutung sich nur noch aus dem jeweiligen Zweck der Geschehnisse nachvollziehen lässt.

Somit können unterschiedliche, formale Reduktionsstrategien ([Mittelstraß und Blasche 1980-1996], siehe unter Begriff „Zustand“) ausgemacht werden, die es zu bewerten gilt:

- Ereignisse als Zustandspaare und
- Zustände als Ereignispaare.

In der geschehnisbezogenen Beschreibung liegt das Gewicht der Beschreibung auf der Transition und im zustandsbezogenen Fall auf der Beschreibung der Zustände. Um eine möglichst ausführliche Beschreibung zu bekommen, könnte man beide Beschreibungen auch kombinieren. Dabei tritt jedoch das Problem auf, dass Sachverhalte unter Umständen mehrmals beschrieben werden und diese Beschreibungen konsistent gehalten werden müssen. I.d.R. wird deswegen nur einer der beiden Fälle verfolgt, oder es werden Mischformen angestrebt, in denen die Zustands- und Transitionsaussageformen keine bzw. geringe Überdeckungen (Befolgung des Orthogonalitätsprinzip) aufweisen.

In der Sprache EPK (siehe 4.2.2) stehen beide Aspekte gleichberechtigt nebeneinander. So wird eine Funktion (Geschehnis) z.B. mit der Bezeichnung „Auftrag eröffnen“ und das

folgende Ereignis (Zustand) mit „Auftrag ist eröffnet“ bezeichnet. Die Redundanz in der Beschreibung ist offensichtlich.

In der Phase der Anforderungsanalyse des Software Engineerings wird i. Allg. verlangt, dass man durch die Beschreibung von Sachverhalten keine Implementierungsentscheidungen vorwegnimmt. D.h., die Entscheidung, wie Handlungen mittels Operationen realisiert werden sollen, soll möglichst in die späten Phasen der Softwareentwicklung verschoben werden. Aus diesem Grund fordern Vertreter einer formal fundierten Softwareentwicklung (siehe [Rumpe 1997]) einen zustandsbezogenen Ansatz. Aufgrund der schweren Verständlichkeit ausschließlich deklarativer Beschreibungen und dem eher operationalen Denken (zumindest) der Softwareentwickler setzt sich in der Praxis aber vermutlich der zweite Ansatz durch. So stellen auch die Zustandsdiagramme der sich zunehmend verbreitenden UML einen transitionsbezogenen Ansatz dar.

Für die Beschreibung von Komponenten ist der geschehnisbezogene der naheliegende Ansatz, da man es mit fertigen Softwareprodukten zu tun hat und somit keine allgemeine, von der Realisierung abstrahierende Beschreibung benötigt wird. Da davon ausgegangen werden kann, dass Benutzer eine Komponente schneller anhand operativer als anhand deklarativer Beschreibungen verstehen, wird im Weiteren eine transitionsbezogene Beschreibung von Komponenten bevorzugt.

6.4.4 Abstraktion durch Unterspezifikation

Beschreibungen für Softwareprodukte abstrahieren oft nicht nur von der konkreten Implementierung (Black-Box-Abstrahierung), sondern zusätzlich vom konkreten funktionalen Verhalten. Sie stellen im Verhältnis zur eigentlichen Implementierung und zur vollständigen Spezifikation nur eine *Unterspezifikation* dar. Beispielsweise werden in den EPKn des R/3-Referenzmodells (siehe Abschnitt 4.2.3) keinerlei Berechnungen (z.B. Addition oder Prozentrechnung) formuliert, da keine speziellen Funktionen dafür vorgesehen sind. Berechnungsvorschriften können nur als Annotationen in Form natürlich- oder anderssprachlicher Beschreibungen (Handbücher, Bildschirmmasken) angegeben werden. Entsprechende Sprachen sind aber nicht Bestandteil des formalen EPK-Modells. Die EPK-Sprache ist somit sehr abstrakt, und da sie nicht turing-mächtig ist, kann das Verhalten von Softwarekomponenten nicht vollständig beschrieben werden.

Die in dieser Arbeit zu entwickelnde Sprache soll grundsätzlich turing-mächtig sein, damit alle berechenbaren Komponenten beschrieben werden können. Jedoch soll andererseits die Möglichkeiten zur Unterspezifikation geboten werden. Beispielsweise sollen Objekttypen/Klassen, wenn sie unwichtig erscheinen, als vollkommen abstrakte Dinge betrachtet werden können. Ferner sollten Methoden und Prozeduren nur bis zu einem bestimmten Detaillierungsgrad beschrieben werden müssen. Auswahlentscheidungen (if_then_else) können ebenfalls so abstrakt bleiben, sodass aufgrund der formalen

Beschreibung nicht entschieden werden kann, welcher Zweig in einem konkreten Fall durch die beschriebene Komponente eingeschlagen werden würde[1]. Solche Auswahlentscheidungen sind Nicht-Determinismen, die durch ein Orakel – den Benutzer – aufgelöst werden.

Ähnlich wie bei den EPKn des R/3-Referenzmodells werden Komponenten also nur soweit beschrieben, bis davon ausgegangen werden kann, dass der Leser sie „versteht“ bzw. mittels anderer Hilfsmittel schnell verstehen kann.

6.4.5 IN-, OUT- und INOUT-Parameter

Im Rahmen der Dokumentation von Geschäftsprozessen (siehe 4.2.1) hat sich der Ansatz durchgesetzt, Aufgaben als Handlungsschemata zu betrachten, die bestimmte Objekte als Eingabe, andere als Ausgabe und dritte zur Transformation sowohl als Ein- wie auch als Ausgabe benötigen. Diesem Muster soll in dieser Arbeit ebenfalls gefolgt werden. Dazu werden drei verschiedene Arten von Parametern für Aktionen benötigt: IN-, OUT- und INOUT-Parameter. Aus diesem Grund kann auch im Komponentenmodell auf Funktionen verzichtet werden. Alle Aufrufschnittstellen der Komponenten, die mindestens einen OUT- oder einen INOUT-Parameter besitzen, werden auch als Funktionen betrachtet. Der Funktionswert einer solchen Schnittstelle ist dann ein Tupel aus allen OUT- und INOUT-Parametern.

6.5 Fazit

Im Rahmen des Entwicklungsprozesses komponentenbasierter Anwendungssysteme werden von den Akteuren Komponentenhersteller, Anwendungsarchitekt und Konfigurierer bestimmte Produkte und Beschreibungen erarbeitet, die entsprechend dem nächsten Akteur im Prozess, also Anwendungsarchitekt, Konfigurierer bzw. Anwender, übergeben werden. Die Zusammenhänge der in dieser Arbeit betrachteten Beschreibungen lassen sich durch einfache Gleichungen (siehe Abbildung 6.1) veranschaulichen.

- Die erste Gleichung besagt, dass Komponentenhersteller Anwendungsarchitekten Beschreibungen ihrer Komponenten liefern, die diese durch Angabe eines Konfigurationsraums zu einer Menge sinnvoller Komponentenkonfigurationen (Anwendungssoftwareprodukt) anordnen.

---

[1] Solche Nicht-Determinismen sind in der EPK-Sprache bei jedem (!) XOR-Konnektor vorhanden, da nicht aufgrund der formalen Definition der EPK, sondern nur aufgrund der textuellen Beschreibung der dem XOR-Konnektor vorgeschalteten Funktion entschieden werden kann, welcher XOR-Zweig eingeschlagen werden muss. Diese textuelle Beschreibung ist jedoch nicht formal und somit nicht automatisch interpretierbar.

- Die zweite Gleichung besagt, dass Konfigurierer aus dem Konfigurationsraum eines Anwendungssoftwareproduktes gemäß der Anforderungen eines Anwenders eine Konfiguration (Anwendungssystem) auswählen. Dazu erstellt er eine Menge von Geschäftsprozessen, die die zukünftigen Interaktionen zwischen Komponenten und Benutzern beschreiben.

Anwendungssoftwareprodukt **:=** Komponenten **plus** Konfigurationsraum

Anwendungssystem **:=** Anwendungssoftwareprodukt **plus** Konfiguration

Abbildung 6.1: Zusammenhang der Beschreibungen

Die in den vorherigen Abschnitten erarbeiteten Anforderungen an diese Beschreibungen lassen sich nun, wie in Tabelle 6.1 geschehen, zusammenfassen.

Rolle	Beschreibungen	Anforderungen
Komponentenhersteller	Komponentenbeschreibungen	(Die Anforderungen der Komponentenhersteller werden in dieser Arbeit nicht betrachtet)
Anwendungsarchitekt	Beschreibungen von Konfigurationsräumen	• Kompositionalität • Black-Box-Abstraktion • Abstraktion von konkreten Komponentenmodellen • Interpretation • Geschäftsprozessorientierung
Konfigurierer	Beschreibung einer konkreten (gebundenen) Konfiguration	• Konfiguration (Auswahl) • Geschäftsprozessorientierung • Interpretation
Anwender	(Nutzung einer Konfiguration)	• Geschäftsprozessorientierung • Verständlichkeit

Tabelle 6.1: Anforderungen an Beschreibungen

7 Ein einfaches Komponentenmodell

Zur Beschreibung von Struktur und Verhalten von Softwarekomponenten – wobei auf dem Verhalten das Hauptaugenmerk liegt – wird nun eine einfache Sprache (*Simple Component Description Language*, SCDL) entwickelt, die die wesentlichen im vorigen Kapitel vorgestellten Anforderungen abdeckt. Die Ideen dazu wurden zu großen Teilen aus den Sprachfamilien übernommen, die in den Abschnitten 4.2 bis 4.4 vorgestellt wurden. Die Semantik von SCDL definiert ein einfaches *Komponentenmodell* (siehe auch Abbildung 7.1). Die Definition erfolgt durch eine Übersetzung von SCDL-Komponentenbeschreibungen in den asynchronen π-Kalkül von Pierce (siehe Abschnitt 17.2 im Anhang). Da für diesen Kalkül schon eine Reihe von höheren, sogar objektorientierten Sprachkonstrukten (siehe [Pierce und Turner 1994]) entwickelt wurden, kann auf diese Arbeiten zurückgegriffen werden. Zusätzlich müssen jedoch die speziellen komponentenbasierten Aspekte berücksichtigt werden.

Gemäß der Forderung aus [Szyperski 1998] wird eine Komponente als eine Black Box betrachtet. Dazu müssen die Schnittstellen sowie das abstrakte Verhalten der Komponente beschrieben werden, jedoch werden über die interne Realisierung keine Informationen preisgegeben. Je nach Beschreibungszweck können die Beschreibungen mehr oder weniger konkret das tatsächliche Verhalten einer Komponente beschreiben.

Eine Komponente besteht aus einer Reihe direkter Schnittstellen (Prozeduren), die die initialen Interaktionsschnittstellen zwischen einer Komponente und ihrer Außenwelt darstellen. Eine Komponente kann – muss aber nicht – eine Menge von Klassen beinhalten, die zusätzlich indirekte Schnittstellen der Komponente (Methoden) zur Verfügung stellen. Wenn eine Komponente Klassen beinhaltet, so werden Instanzen (Objekte) dieser Klassen über die direkten Schnittstellen erzeugt und können dann wie in einer objektorientierten Sprache üblich verwaltet werden.

Eine Komponente kann andere Komponenten implizit benutzen, ohne den Import explizit anzugeben. Die Importbeziehungen zwischen verschiedenen Komponenten, wie man sie von Sprachen wie Java[1] und Modula-2 her kennt, lassen sich aus diesen „Benutzt“-Beziehungen indirekt ableiten. Gleiches gilt auch für die ausgehenden Schnittstellen zu importierten Komponenten. Sie sind ebenfalls nur implizit gegeben und lassen sich ebenfalls aus der Gesamtbeschreibung einer Komponente ermitteln. Beschreibungselemente zur Definition von

[1] In Java kann ebenfalls auf die Import-Anweisungen verzichtet werden, indem allen referenzierten Klassen eines Pakets der jeweilige Name des Pakets vorangestellt wird.

öffentlichen und geschützten Methoden und Prozeduren werden nicht benötigt, da diese Informationen Bestandteile der Implementierung sind.

Pro Klasse und Komponente werden die Signaturen der Methoden/Prozeduren beschrieben. Die Parameter einer Signatur werden mit einer Richtung beschriftet (`in`, `inout` oder `out`), die kennzeichnet, ob der Parameter einen Wert verlangt, ändert oder ausgibt. Funktionswerte werden so z.B. durch *`OUT`-Parameter* (Ausgabeparameter) dargestellt. Sowohl das Verhalten der Klassen als auch das der gesamten Komponente werden über Prozessterme beschrieben.

Spezielle Komponenten, sogenannte *Komponenten-Frameworks*, zeichnen sich dadurch aus, dass sie durch andere Komponenten parametriert werden müssen. Frameworks unterscheiden sich von „normalen“ Komponenten – sogenannten *geschlossenen Komponenten* – dadurch, dass sie zusätzlich zu den Schnittstellen und den Verhaltensbeschreibungen sogenannte Slots besitzen, in die andere Komponenten eingesetzt werden können. Ein *Slot* ist ein Steckplatz, der durch einen Kontrakt typisiert wird. Der Kontrakt muss durch eine eingesetzte Komponente eingehalten werden. Er wird genauso wie eine einfache Komponente durch seine Schnittstellen und sein Verhalten beschrieben.

Durch das Einsetzen von atomaren Komponenten in Komponenten-Frameworks entstehen *komplexe Komponenten*, die auch wieder in Komponenten-Frameworks eingesetzt werden können. Stellen komplexe Komponenten für den Endanwender eigenständig nutzbare Produkte dar, so wird auch von Anwendungssoftwareprodukten gesprochen. Diese werden vor ihrem Einsatz zu konkreten *Anwendungssystemen* konfiguriert. In Abbildung 7.1 werden diese Sachverhalte nochmals veranschaulicht. Die Zusammenhänge zwischen atomaren Komponenten und Komponenten-Frameworks werden nun in diesem Kapitel genauer definiert. Komplexe Komponenten und ihre Komposition und Konfiguration zu Anwendungssystemen werden in den nachfolgenden Kapiteln behandelt.

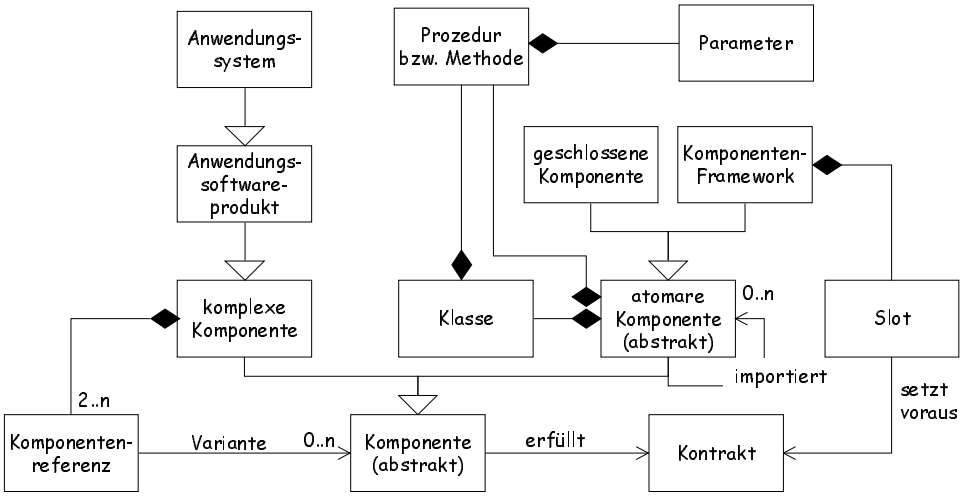


Abbildung 7.1: Komponentenmodell

Zuerst wird nun in Abschnitt 7.1 ein kleines SCDL-Beispiel gegeben, das zur Motivation und zum Grundverständnis von SCDL beitragen soll. In Abschnitt 7.2 wird dann die Syntax und in Abschnitt 7.3 die Semantik von SCDL definiert. In Abschnitt 7.4 werden Interaktionssequenzen (Traces) eingeführt, die im weiteren Verlauf der Arbeit zur Erzeugung von Geschäftsprozessen benötigt werden.

7.1 Ein SCDL-Beispiel

Bevor nun mit der Definition von Syntax und Semantik von SCDL begonnen wird, sollen die wesentlichen Konstrukte der Sprache anhand eines Beispiels erklärt werden. In diesem Beispiel wird ein kleines Client/Server-System vorgestellt, das auf einem Framework mit einem vordefiniertem Verhalten basiert. Der Framework fordert von einem Server einen Dienst (Service) an und stellt diesen einem Client zur Verfügung. Der Framework muss dazu mit je einer Client- und einer Server-Komponente parametriert werden. Voraussetzung ist jedoch, dass diese Komponenten jeweils vorgegebenen Kontrakten – also Schnittstellen- und Verhaltensvereinbarungen – entsprechen.

Im Beispiel (siehe Abbildung 7.2) werden zunächst die zwei Kontrakte `serverC` und `clientC` definiert, die jeweils Schnittstellen und Verhalten eines eigenständigen Typs von Komponenten definieren. Komponenten eines solchen Typs haben eine Reihe von direkten sowie zusätzliche, indirekte (objektorientierte) Schnittstellen. Zur Veranschaulichung soll der Kontrakt `clientC` dienen. Eine Komponente, die diesem Kontrakt entspricht, muss eine direkte Prozedur `createClient` anbieten, die ein Objekt zurückgibt, das der Definition eines Clients (`Client`) entspricht. Ein solches `Client`-Objekt, das als sogenannter *Objekttyp* (`objecttype`) beschrieben wird, unterstützt den Aufruf einer Methode mit dem Namen `use`, der ein `Service`-Objekt übergeben wird. Dieses `Service`-Objekt ist in dem Kontrakt `serverC` als Objekttyp definiert.

```
/****************** KONTRAKTE *************************/
contract serverC {
     objecttype Service {
          serve();
     }
     createService(out s:serverC.Service)
          = new(serverC.Service s); return(s)

     initialized = createService?(s);initialized.
}

contract clientC {
     objecttype Client {
          use(in s:serverC.Service) = s.serve!(); return

          initialized = use?(s); initialized.
     }
     createClient(out c:clientC.Client) = new(clientC.Client c);

     initialized = createClient?(c);initialized.
}

/****************** KOMPONENTEN ************************/
component csFramework {
     slot serverC s;
     slot clientC c;

     initialized = s.createService!(sv); c.createClient!(cl);
cl.use!(sv).
}

component server {
     class Service {
          serve();
     }
     createService(out s:server.Service)
               = new(server.Service s); return(s)

     initialized = createService?(s);initialized.
}

component client {
     class Client {
          ...
          /* ähnlich wie clientC */
     }
}

complex asp = csFramework(server, client);
```

Abbildung 7.2: Beispiel für eine SCDL-Beschreibung

Damit sind die statischen Aspekte des Kontraktes beschrieben. Zusätzlich wird das Verhalten der Komponente (direkte Ebene) sowie der Objekte des Objekttyps `Client` (indirekte Ebene) beschrieben. Eine solche Verhaltensbeschreibung wird in beiden Fällen mit dem Schlüsselwort `initialized` eingeleitet. Das Verhalten wird durch Zustände und Transitionen, die Wechsel zwischen Zuständen definieren, beschrieben. Der Zustand einer Komponente kann als ein Prozess verstanden werden, der auf eine oder mehrere alternative Interaktionen mit anderen Komponenten wartet. Tritt eine Aktion ein, so wird die Transition durchgeführt und die Komponente wechselt in einen Folgezustand. So kann z.B. die direkte Prozedur `createClient` einer Komponente, die dem Kontrakt `clientC` entsprechen soll, beliebig oft nacheinander aufgerufen werden.

Diese Tatsache ist aus der Verhaltensbeschreibung unmittelbar ersichtlich: Der Anfangszustand der Komponente wird durch den Prozess `initialized` beschrieben. In diesem Zustand wartet eine solche Komponente auf den Aufruf der Prozedur `createClient`, was durch das Fragezeichen („?“) hinter dem Prozedurnamen kenntlich gemacht wird. Nach einem Aufruf wechselt die Komponente sofort wieder in den ursprünglichen Zustand `initialized`. Durch `createClient` wird ein neues Objekt vom Objekttyp `Client` erzeugt. Ein solches Objekt muss mindestens folgendes Verhalten aufweisen: Zuerst muss die Methode `use` aufgerufen werden können. In dieser Methode wird daraufhin von dem aktuellen Parameter `s` – ein Objekt vom Objekttyp `Service` – die Methode `serve` aufgerufen. Danach kann das Objekt wieder einen Aufruf der Methode `use` akzeptieren usw. Methoden- und Prozeduraufrufe werden also durch das Ausrufungszeichen („!“) kenntlich gemacht. Ähnlich wie der Kontrakt `clientC` ist auch der Kontrakt `serverC` aufgebaut. Er wird hier nicht näher beschrieben.

Der oben erwähnte Framework – im Beispiel heißt er `csFramework`– ist kein Kontrakt, sondern eine parametrierbare Komponente (`component`). Sie hat als Besonderheit zwei Steckplätze (Slots), an denen sie erweitert (parametriert) werden muss. Solche erweiterbaren Komponenten werden als *Komponenten-Frameworks* bezeichnet. Diese Komponente muss durch jeweils eine Komponente `c` vom Typ `clientC` bzw. `s` vom Typ `serverC` parametriert werden und wird erst dadurch zu einer komplexen Komponente, die auch tatsächlich ausführbar ist. Da `csFramework` in verschiedenen Zusammenhängen unterschiedlich parametriert werden kann, können folglich verschiedene komplexe Komponenten aus einem Komponenten-Framework abgeleitet werden.

Die Komponente `csFramework` besitzt keinerlei direkte oder indirekte Schnittstellen. Das Verhalten dieser Komponente ist folgendermaßen definiert: Zuerst ruft sie die Prozedur `createService` der Komponente `s` auf, die einen `Service` zurückgibt. Danach wird die Komponente `c` aufgerufen, die einen `Client` zurückgibt. Zum Schluss wird dem `Client` über die Methode `use` der Dienst `Service` übergeben. Natürlich kann eine Komponente mit

Slots (Komponenten-Framework) auch eigene direkte und indirekte Schnittstellen besitzen und diese nach außen hin bekannt geben. Indirekte Schnittstellen werden innerhalb von Klassen (Schlüsselwort `class`) definiert. Während Komponenten Realisierungen entsprechender Kontrakte sind, sind Klassen also die Realisierungen entsprechender Objekttypen.

Im Beispiel wurden zwei Komponenten (`server` und `client`) definiert, die die beiden vorgestellten Kontrakte `serverC` und `clientC` erfüllen sollen. Ihre Beschreibungen sind nur Kopien der Kontrakte. Im Beispiel ist weiterhin eine komplexe Komponente (`asp`) definiert. Zur Definition werden alle drei bisher vorgestellten Komponenten importiert, und die zwei Slots des Komponenten-Frameworks `csFramework` werden mit den Komponenten `server` und `client` parametriert. Dadurch entsteht die neue konkrete Komponente `asp`.

In Abbildung 7.3 ist das Beispielsystem in Form einer Schichtenarchitektur dargestellt. Die Schichten geben die Abhängigkeiten der Komponenten zueinander an. Diese Abhängigkeiten werden in modularen Programmiersprachen wie Java oder Modula-2 meist explizit durch Importanweisungen definiert. In SCDL werden der Einfachheit halber die importierten Komponenten global referenziert, dadurch kann auf diese Anweisungen verzichtet werden. Die Parameterbeziehungen – also die Informationen darüber, welche Komponente der Konfiguration welche andere Komponente als Parameter erhält – sind horizontal durch kleine schwarze Pfeile angedeutet.

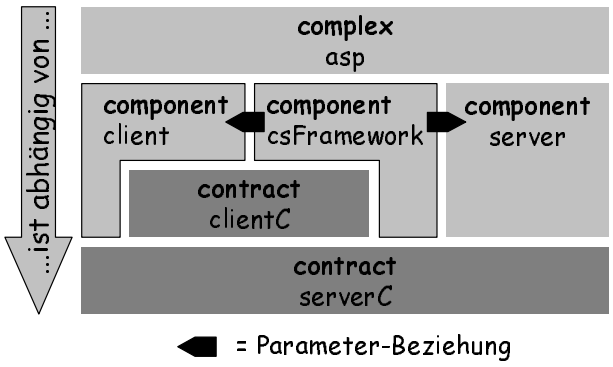


Abbildung 7.3: Schichtenarchitektur zum Beispiel

Die operationale Semantik der Komponentenbeschreibung wird wesentlich durch die Verhaltensbeschreibungen (Schlüsselwort `initialized`), die wiederum auf den Signaturen der direkten (Prozeduren) und der indirekten Schnittstellen (Methoden) basieren, definiert. Durch sie werden die erlaubten Aufrufreihenfolgen (Traces, Sequenzen) der Prozeduren und Methoden einer Komponente festgelegt.

Auffallend ist, dass es kein Main-Programm oder eine initiale, ausgezeichnete Methode gibt, über die das beschriebene System gestartet wird. Statt dessen werden in SCDL alle

Komponenten in Definitionsreihenfolge durch Aufruf der Verhaltensbeschreibung (`initialized`) sofort initialisiert. Sie beschreiben somit parallele, eigenständige Prozesse.

Komponenten werden durch Import oder durch Parametrierung einander bekannt gemacht werden. Die Importabhängigkeiten dürfen nicht zyklisch sein[1]. Zyklische Beziehungen zwischen Komponenten können jedoch durch die Parametrierung zur Konfigurationszeit erreicht werden.

Die parallel initialisierten Komponenten können von anderen Komponenten aufgerufen werden bzw. können selbst andere Komponenten aufrufen. So ruft z.B. die komplexe Komponente `asp` die Prozedur `createService` der als Parameter übergebenen Komponente `s` auf. Da diese Komponente (als `server`) in der Komponente `asp` ebenfalls importiert wurde und gemäß ihrer Verhaltensbeschreibung auf diesen Befehl wartet, können die beiden Komponenten miteinander kommunizieren.

7.2 Syntax von SCDL

Die Syntax von SCDL wird nun in EBNF-Notation (siehe Abschnitt 16.2) angegeben. Eine Beschreibung eines Komponentensystems auf Grundlage der Sprache SCDL besteht aus einer Reihe von Kontrakt- und Komponentendefinitionen sowie einer Reihe von Definitionen komplexer Komponenten, die Komposita atomarer und/oder komplexer Subkomponenten darstellen. Ein Kompositum von Komponenten eines Anwendungssoftwareprodukts (siehe Kapitel 9) wird also ausschließlich durch `complex`-Anweisungen realisiert.

Kontrakte und atomare Komponenten unterscheiden sich syntaktisch kaum. Lediglich die Schlüsselwörter `component` und `class` der Komponente müssen beim Kontrakt durch die Schlüsselwörter `contract` bzw. `objecttype` ersetzt werden. Wegen dieser Ähnlichkeit wird in diesem Abschnitt auf die formale Syntaxdefinition von Kontrakten verzichtet.

```
<SYSTEM>  ::= {(<CONTRACT> | <COMPONENT> | <COMPLEX-COMP>)}+.
```

Komplexe Komponenten bestehen aus einem Komponenten-Framework (`PATH`) und einer Reihe von Komponenten, die als Parameter (Parameterkomponenten), in die Slots des Frameworks eingesetzt werden. Somit werden für eine komplexe Komponente immer einfachere Komponenten benötigt.

---

[1] Wie in [Szyperski 1998] gezeigt wird, ergeben zyklische Abhängigkeiten i.d.R. keinen Sinn, da alle Komponenten, die Teil des Zyklus sind, nicht ohne die anderen Komponenten eingesetzt werden können. Sie müssen immer zusammen entwickelt werden und sind in diesem Sinne naturgemäß keine eigenständigen Komponenten.

Die komplexe Komponente wird durch einen Pfadausdruck (PATH) eindeutig identifiziert.

```
<COMPLEX-COMP>      ::= complex <PATH>=<PATH> <SLOTCOMPONENTS>;.
<SLOTCOMPONENTS>    ::= [ "(" <PATH> {, <PATH>} ")" ].
```

Eine Komponente bzw. ein Kontrakt besteht aus

- einem Identifizierer (PATH), der ähnlich wie ein Paketname unter Java (siehe [Flanagan 1997]) eindeutig aus mehreren IDs– jeweils durch einen Punkt („.“) getrennt – als Pfad aufgebaut wird,
- einer optionalen Liste von Slots, die die Parametrierungsmöglichkeiten einer Komponente (genauer: eines Komponenten-Frameworks) mit anderen Komponenten beschreibt, wobei ein Slot aus dem Namen des Kontraktes, der von einer einzusetzenden Komponente unterstützt werden muss, und einem Variablennamen, über den die Komponente referenziert wird, besteht,
- einer Menge von Klassen (CLASSES, bzw. beim Kontrakt: von Objekttypen, (OBJECTTYPES)), in denen die indirekten Schnittstellen einer Komponente definiert werden,
- einer Menge von (direkten) Prozeduren (SIGS), die einen weiteren Teil der funktionalen Eigenschaften einer Komponente beschreiben, und
- der Verhaltensdefinition (BEHAVIOUR), die wiederum aus einer Reihe von kombinierten Zustands-/Transitionsbeschreibungen (STDESCRIPTIONS) inklusive eines initialen Zustands (initialized) besteht, wobei die Transitionsbeschreibungen definieren, welche Prozeduren der Komponente wann und unter welchen Bedingungen aufgerufen werden können.

```
<COMPONENT>         ::= component <PATH> "{" <SLOTS> <CLASSES>
                                <SIGS> [ <BEHAVIOUR> ]"}".
<PATH>              ::= <ID> | <ID>"."<PATH>.
<SLOTS>             ::= { slot <PATH> <ID>; }.
<CLASSES>           ::= { <CLASS> }.
<SIGS>              ::= { <PROCEDURE> }.
<BEHAVIOUR>         ::= initialized = <TRANSITION>"." [ <STDESCRIPTIONS> ].
<STDESCRIPTIONS>    ::= <STATE> = <TRANSITION>"." [ <STDESCRIPTIONS> ].
<STATE>             ::= <ID>  |<ID>"("<FPARA>")".
```

Eine Klasse ist ähnlich aufgebaut wie eine Komponente. Sie besitzt ebenfalls

- einen (innerhalb der Komponente) eindeutigen Identifizierer (ID),
- eine Reihe von Methoden (entsprechen den indirekten Prozeduren, ISIGS),
- und eine Verhaltensbeschreibung (BEHAVIOUR).

Wesentlicher Unterschied zwischen Komponenten und Klassen ist, dass zur Laufzeit eines Komponentensystems grundsätzlich beliebig viele Instanzen einer Klasse (= Objekte) erzeugt werden können, während von Komponenten immer nur eine Instanz existiert.

```
<CLASS>       ::=  class <ID> "{" <ISIGS> [ <BEHAVIOUR> ] "}".
<ISIGS>       ::=  { <METHOD> }.
<METHOD>      ::=  <PROCEDURE>.
```

Sowohl Prozeduren als auch Methoden werden innerhalb ihrer Komponente bzw. innerhalb ihrer Klasse eindeutig durch einen Bezeichner (`ID`) identifiziert. Beide besitzen eine Liste formaler Parameter (`FPARAS`), die beim Aufruf durch aktuelle Parameter belegt werden. Die formalen Parameter sind typisiert und besitzen eine Richtungsangabe (`in`, `inout` oder `out`). Durch die Richtung wird angegeben, ob ein Parameter der Prozedur/Methode als Argument übergeben wird (`in`), ob der Parameter einen Ergebnisparameter beschreibt (`out`) oder beide Eigenschaften besitzt (`inout`).

Eine Prozedur und auch eine Methode – auf diese Unterscheidung wird im Weiteren, wenn nicht unbedingt notwendig, verzichtet – kann in ihrer Funktionalität genauer beschrieben werden. Dazu wird ein Prozess (`PROCBODY`) definiert. Entfällt diese Definition, wird eine Standardfunktionalität angenommen, die im Abschnitt 7.3.16 definiert wird. Die Funktionalität kann durch den Aufruf anderer Prozeduren und durch das Erzeugen neuer Objekte (`new`) genauer beschrieben werden. Eine Prozedur wird mit dem Return-Befehl abgeschlossen, dem alle aktuellen Parameter übergeben werden, die als Ergebnisparameter (`INOUT`- und `OUT`-Parameter) im Kopf der Prozedur definiert wurden und somit zurückgegeben werden müssen.

```
<PROCEDURE>         ::= <ID> "(" [<FPARAS>] ")" ( ; | = <PROCBODY> ).
<FPARAS>            ::= <FPARA> | <FPARA>, <FPARAS>.
<FPARA>             ::= (in | out | inout) <ID> : <PATH>.
<PROCBODY>          ::= <ACTION>;<PROCBODY>
                    |   <NEW>;<PROCBODY>
                    |   <RETURN>.
<NEW>               ::= new "(" <PATH> <ID> ")".
<RETURN>            ::= return "(" [<APARAS>] ")".
```

Die Verhaltensbeschreibungen einer Komponente oder einer Klasse sind mit den Zuständen und Transitionen einer Zustandsmaschine (vgl. Statecharts, Abschnitt 4.3.2) vergleichbar. Die Transitionen werden in einer Schreibweise formuliert, die an einen Prozess des asynchronen π-Kalküls Pict (siehe Abschnitt 17.2) erinnert. Eine Transition kann entweder leer sein („`()`“), aus einer Aktion (Aufrufen oder Aufgerufen-werden) und einer Resttransition, aus einer *Auswahl* (siehe Abschnitte 17.2.6 und 17.2.7) oder einer parallelen Komposition von anderen Transitionen oder aus einem Wechsel zu einem anderen Zustand bestehen. Jede Transition endet entweder mit dem leeren Prozess – das bedeutet, dass die Komponente/Klasse anhält – oder mit dem Übergang zu einem neuen Zustand (`STATECHANGE`).

```
<TRANSITION>        ::= “()”
                    |   <ACTION>;<TRANSITION>
                    |   <ECARULE> { + <ECARULE> }+
                    |   “(” <TRANSITION> { “|” <TRANSITION> }+ “)”
                    |   <STATECHANGE>.
<STATECHANGE>       ::= <ID>!“(” [<APARAS>] “)”.
```

Die Bestandteile der Auswahl sind einzelne *ECA-Regeln*, die aus einer Ereignisbeschreibung, einer Bedingung und einer Resttransition bestehen.

```
<ECARULE>           ::= on <EVENT> [ if <CONDITION> ] do <TRANSITION>.
<EVENT>             ::= <RECV>.
<CONDITION>         ::= <PATH>!“(” [<APARAS>] “)”.
```

Die elementaren Aktionen von SCDL sind das Aufrufen („!“) und das Aufgerufen-Werden („?“) von Prozeduren/Methoden. Ein Aufruf in SCDL ist dabei immer synchron, d.h., der Sender wartet solange, bis der Empfängerprozess die zugehörige Methode abgearbeitet hat und die berechneten Parameter (`inout` und `out`) zurückgegeben hat.

```
<ACTION>            ::= <RECV>
                    |   <SEND>.
<RECV>              ::=   <ID>?“(” [<APARAS>] “)”.
<SEND>              ::= <PATH>!“(” [<APARAS>] “)”.
<APARAS>            ::= <APARA> | <APARA>, <APARAS>.
<APARA>             ::= <ID>
```

7.3 Semantik von SCDL

Die formale Semantik der Komponentenbeschreibungssprache SCDL wird durch eine Übersetzung in die Sprache Pict angeben. Diese Sprache zeichnet sich durch ihren grundlegend prozessorientierten Charakter (siehe Kapitel 17 in Teil V) aus. Prozessterme in Pict setzen sich aus elementaren Sende- und Empfangsaktionen zusammen, die hintereinander, alternativ oder parallel ausgeführt werden dürfen. Sende- und Empfangsaktionen zweier Prozessterme können interagieren, wenn sie den gleichen Kommunikationskanal benutzen. Kommunikationskanäle können beliebig erzeugt werden. Nicht-Determinismus ist u.a. dadurch gegeben, dass mehrere Sender oder Empfänger den gleichen Kanal benutzen und keine Reihenfolge definiert ist, wann welcher Prozess auf den Kanal zugreift. Alle komplexeren Sprachkonstrukte wie abstrakte Datentypen, Variablen und objektorientierte Elemente lassen sich – wie so oft bei formalen Sprachen – auf den grundlegenden Kalkül zurückführen. Hierzu kann auf eine Reihe wichtiger Arbeiten verwiesen werden, die mit denen von Milner et al. ([Milner 1989], [Milner, Parrow und Walker 1992a], [Milner, Parrow und Walker 1992b]) ihren Anfang nahm und dann von Pierce, Turner (siehe [Pierce und Turner 1994], [Pierce und Turner 1997]) und Walker (siehe [Walker 1995]) in Richtung Objektorientierung fortgeführt wurde. Die Stärke dieser

prozessorientierten Sprachen liegt in ihrem formalen Fundament, das zur Fundierung von SCDL konsequent genutzt wird.

Neben der Formalität zeichnen sich Prozesskalküle wie der π-Kalkül durch ihre Kompositionalität aus. Durch die Basisoperatoren wie sequentielle, parallele und alternative Komposition können sehr einfach neue, komplexere Terme aufgebaut werden, deren Semantik dann zugleich durch diese Operatoren definiert ist.

Die gesamte Übersetzersemantik wurde im Rahmen der Arbeit in Form von Transformationsregeln erarbeitet, um später (siehe Kapitel 11) direkt in einem Compiler umgesetzt werden zu können.

7.3.1 Übersetzung eines Komponentensystems

Ein Komponentensystem bzw. die SCDL-Beschreibung dieses Systems besteht aus einer Reihe von Kontrakten sowie mehreren Komponenten, zu denen auch komplexe Komponenten gehören, die aus einfacheren Komponenten zusammengesetzt sind.

Kontrakte werden – unter Berücksichtigung der syntaktischen Unterschiede (siehe Abschnitt 7.2) – genauso wie Komponenten übersetzt. Sie haben jedoch aus Sicht der Semantik nur insofern eine Bedeutung, wie sie zur Überprüfung der Kompatibilität verschiedener Komponenten herangezogen werden sollen. So kann beispielsweise zum Übersetzungszeitpunkt unter bestimmten Voraussetzungen „statisch“ geprüft werden, ob die Aufrufparameter zur Laufzeit immer vom richtigen Typ sind. Dieser Gedanke wird in dieser Arbeit nicht weiter verfolgt. Vielmehr soll von vornherein zugesichert werden, dass die entsprechenden Elemente (Komponenten, Klassen, Methoden usw.) zueinander kompatibel sind. So kann auf die explizite Überprüfung verzichtet werden. Die Zusicherungen werden von den Komponentenherstellern und/oder Anwendungsarchitekten gegeben. Welche Komponente in welchem Fall einsetzbar (substituierbar) ist, muss dazu (siehe Kapitel 9) im Rahmen der Beschreibung gültiger Konfigurationen eines Anwendungssoftwareprodukts explizit angegeben werden. Erst bei der Definition dieses Konfigurationsraums muss berücksichtigt werden, welche Komponente mit welcher anderen Komponente wirklich parametriert werden darf.

Ein vollständiges *Kompositum* eines Anwendungssoftwareprodukts[1] eines Komponentensystems wird durch eine Reihe von `complex`-Anweisungen definiert, die

[1] Ein Kompositum von Komponenten eines Anwendungssoftwareprodukts werden aus Sicht der Konfiguration als *gebundene Konfiguration* (siehe Abschnitt 9.1) verstanden, also als eine Konfiguration, in der keine Komponenten mehr ausgewählt werden müssen, um sie als Anwendungssystem einsetzen zu können.

angeben, welche Komponenten-Frameworks mit welchen anderen Komponenten „bestückt“, d.h. parametriert werden sollen. Wie diese nach Pict übersetzt werden, wird in Abschnitt 7.3.10 erklärt.

Im Folgenden braucht also nur noch die Übersetzung von atomaren und komplexen Komponenten betrachtet werden, da Kontrakte wie Komponenten übersetzt werden. Die Transformationsregeln werden dazu „top-down“ dargestellt. Da eine Komponente aus mehreren Einzelteilen besteht, ergibt sich die Übersetzung einer Komponente aus der Übersetzung ihrer Einzelteile. In den Regeln werden die Bestandteile als Variablen (= Nicht-Terminale der Syntaxdefinition, siehe Abschnitt 16.2) in spitzen Klammern („<“ und „>“) angegeben. Sie werden entweder, wenn es sich um Zeichenketten handelt (wie bei `<ID>` oder `<PATH>`), direkt zur Einsetzung unter Pict genutzt oder als Argumente an weitere Transformationsregeln übergeben.

Die Namen der Regeln beginnen immer mit dem Präfix „`trans`“. Einige Regeln werden mit einem Namen, der sich aus dem Plural seiner Bestandteile ergibt (z.B. `transClasses`), referenziert. Jedoch wird die Regel nur im Singular (z.B. `transClass`), also nur für die einfache Anwendung definiert. Für eine vollständige Definition und Implementierung eines Übersetzers müssten diese Regeln dann mehrfach, z.B. rekursiv angewandt werden.

7.3.2 Übersetzung einer Komponente: `transComponent`

Eine Komponente wird nach Pict übersetzt, indem

- ihre Klassen übersetzt,
- aus den Signaturen ihrer Prozeduren Pict-Type-Definitionen abgeleitet,
- aus den Zustands-/Transitionsbeschreibungen der Komponente ein Reihe globaler Pict-Prozessagenten, die das Verhalten der Komponente repräsentieren, generiert und
- aus den Prozeduren jeweils lokale Pict-Prozessagenten erzeugt werden.

```
transComponent(component <PATH>{<SLOTS> <CLASSES> <SIGS> <BEHAVIOUR> }
     :=
     transClasses(<CLASSES>)
     transSignatures(<PATH>, <SIGS>)
     transComponentAgent(<PATH>, <SLOTS>, <SIGS>, <BEHAVIOUR>)
```

7.3.3 Übersetzung von Klassen: `transClass`

Eine Klasse wird nach Pict übersetzt, indem

- aus den Signaturen ihrer Methoden Pict-Type-Definitionen abgeleitet,
- aus den Zustands-/Transitionsbeschreibungen jeder Klasse eine Reihe Pict-Prozessagenten, die das Verhalten der Klasse repräsentieren, generiert und

- aus den Methoden jeweils lokale Pict-Prozessagenten erzeugt werden.

```
transClass(class <ID> { <ISIGS> <BEHAVIOUR> } }
     :=
     transSignatures(<ID>, <ISIGS>)
     transClassAgent(<ID>, <ISIGS>, <BEHAVIOUR>)
```

7.3.4 Übersetzung der Signaturen von Komponenten und Klassen: `transSignatures`

Die Signaturen der Prozeduren bzw. Methoden der Komponenten bzw. Klassen werden in Pict in einem Tupel zusammengefasst und als Pict-Typ definiert. Ein solcher Typ beschreibt ein Tupel aus Kanälen („**^**“). Jeder Kanal dient der Repräsentation einer Prozedur/Methode. Wird z.B. eine Komponente <PATH> definiert, so bedeutet das in Pict, dass ein Tupel vom Pict-Typ <PATH> erzeugt wird, in dem pro Prozedur ein Kanal definiert ist, über den auf die Funktionalität zugegriffen werden kann.

```
transSignatures( <PATH>, <ID1>( <FPARAS1> ) = ...
                         ...
                       <IDn>( <FPARASn> ) = ...
               )
     :=
     type <PATH> =
          [
               <ID1> = ^[ transChannelType(<FPARAS1>) ]
                                ...
               <IDn> = ^[ transChannelType(<FPARASn>) ]
          ]
```

7.3.5 Übersetzung der Signatur einer Prozedur/Methode: `transChannelType`

Die Parameter einer Prozedur (und Methode) teilen sich in `IN`-, `INOUT`- und `OUT`-Parameter. Die `IN`- und `INOUT`-Parameter (Fall 1) werden der Prozedur übergeben, die `INOUT`- und `OUT`-Parameter (Fall 2) werden von der Prozedur an die Aufrufer zurückgegeben. Der erste Fall wird wie in Pict üblich durch eine Liste von Parametern realisiert, die in einem Tupel zusammengefasst und dem Prozessagenten der Prozedur übergeben werden. Für den zweiten Fall muss zusätzlich in der Liste der Parameter ein *Return-Kanal* („**^[...]**“, siehe Abschnitt 17.2) spezifiziert werden, auf dem die entsprechenden Parameter zurückgegeben werden. Da Prozeduraufrufe in SCDL synchronisiert ablaufen sollen, kann der Return-Kanal gleichzeitig zur Synchronisation genutzt werden. Deshalb wird er auch dann benötigt, wenn eine Prozedur keine `INOUT`- oder `OUT`-Parameter enthält. Der Return-Kanal ist dann ein Kanal, auf dem leere Tupel verschickt werden.

Es seien <D_1> <T_1> <P_1>, ..., <D_n> <T_n> <P_n> die formalen Parameter der Prozedur <PATH> (`D` steht hier für die Richtung (`in`, `inout` oder `out`), `T` für den Typ und `P` für den Namen des Parameters). Aus den Typen werden nun zwei Listen aufgebaut: eine

Input-Liste (<T_{i_1}> <T_{i_2}> ...<T_{i_m}>) mit allen IN- und INOUT-Parametern und eine Output-Liste (<$T_{i_{m+1}}$> ... <T_{i_o}>) mit allen INOUT- und OUT-Parametern. Für die beiden Listen gilt weiterhin[1]:

$$
\begin{aligned}
& m \in \{1,\ldots,n\} \\
\wedge\quad & o \in \mathbb{N} \\
\wedge\quad & i_1,\ldots,i_m,i_{m+1},\ldots,i_o \in \{1,\ldots,n\} \\
\wedge\quad & i_1<\ldots<i_m \\
\wedge\quad & i_{m+1}<\ldots<i_o \\
\wedge\quad & Di_1, \ldots, Di_m \in \{\text{in}, \text{inout}\} \\
\wedge\quad & Di_{m+1}, \ldots, Di_o \in \{\text{inout}, \text{out}\}
\end{aligned}
$$

Dann kann die Übersetzung durch folgende Regel angegeben werden:

transChannelType(<D_1> <T_1> <P_1>, ..., <D_n> <T_n> <P_n>)
:= <Ti_1> <Ti_2> ... <Ti_m> **^[**<Ti_{m+1}> ... <Ti_o>**]**

7.3.6 Übersetzung der Verhaltensbeschreibung und der Prozeduren einer Komponente: transComponentAgent

Eine Komponentenbeschreibung <PATH> wird in Pict in einem Prozessagenten new_<PATH> zusammengefasst. Eine (einzige!) Komponenteninstanz (siehe Abschnitt 3.2.9) entsteht zur Laufzeit durch den Aufruf dieses Prozessagenten und wird über einen Return-Kanal (this) zurückgegeben. Dieser Aufruf wird bei geschlossenen Komponenten in der Übersetzung unmittelbar hinter der Definition des Prozessagenten angegeben, d.h., eine geschlossene Komponente wird, sobald sie Teil einer Beschreibung ist, auch initialisiert. Bei Komponenten-Frameworks wird dieser Aufruf aus der complex-Anweisung generiert (siehe Abschnitt 7.2). Zur Laufzeit wird die Komponenteninstanz der Komponente <PATH> als Tupel mit dem gleichen Namen (<PATH>) repräsentiert.

Parametrierbare Komponenten (Komponenten-Frameworks) besitzen Slots. Diese Slots werden als formale Parameter des Prozessagenten definiert.

```
transComponentAgent(<PATH>, <SLOTS>, <SIGS>, <BEHAVIOUR>)
    :=
    def new_<PATH> [ transSlotSigs(<SLOTS>) this:^<PATH> ] =
            transProcCalls(<SIGS>)
            transProcedures(<SIGS>)
            transBehaviour(<BEHAVIOUR>)
            transThisStructur(<SIGS>)
```

[1] Die Indizes $i_1,\ldots,i_n$ kennzeichnen die Positionen der Parameter in einer SCDL-Signatur. Grundsätzlich müssen die Parameter in der Übersetzung in der gleichen Reihenfolge bleiben.

7.3.7 Übersetzung der Verhaltensbeschreibung und der Prozeduren einer Klasse: `transClassAgent`

Ähnlich wie die Komponenten selbst wird auch jede Klasse einer Komponente in einen Prozessagenten in Pict übersetzt. So kann bei der Übersetzung auch das gleiche Schema wie in Abschnitt 7.3.6 angewendet werden. Der einzige Unterschied besteht darin, dass Klassen keine Slots besitzen. Die Liste der Parameter der zu erzeugenden Agenten besteht somit nur aus dem Rückgabeparameter (`this`), über den die erzeugte Struktur zurückgegeben wird, die eine Instanz einer Klasse zur Laufzeit repräsentiert.

```
transClassAgent(<ID>, <SIGS>, <BEHAVIOUR>)
     :=
     def new_<ID> [ this:^<ID> ] =
               transProcCalls(<SIGS>)
               transProcedures(<SIGS>)
               transBehaviour(<BEHAVIOUR>)
               transThisStructur(<SIGS>)
```

7.3.8 Übersetzung der Rückgabestruktur für die Klassen- und Komponentenagenten: `transThisStructur`

Jede Komponente und jede Klasse wird zur Laufzeit durch eine Pict-Struktur repräsentiert, die durch einen Komponenten- bzw. Klassenagenten erzeugt und auf dem Returnkanal `this` zurückgegeben wird. Der Prozessagent `transThisStructur` definiert die Rückgabe dieser Struktur über den Kanal `this`.

```
transThisStructur( procedure <P1> ...
                     ...
                   procedure <PN> ... ) =
     this![ <P1> = <P1>
             ...
            <PN> = <PN> ]
```

7.3.9 Übersetzung der Slots: `transSlotSigs`

Pro Slot `<C>` eines Typs `<T>` eines Komponenten-Frameworks wird je ein Parameter `<C>` des Pict-Typs `<T>` definiert, über den dann die Funktionalität der Komponente abrufbar ist.

```
transSlotSig( slot <T> <C> ) := <C>:<T>
```

7.3.10 Übersetzung der Subkomponentendefinitionen: `transComplexComps`

Komplexe Komponenten können als vollständig vorgegebene Komponenten in anderen Komponenten referenziert werden. Dazu werden sie neben den atomaren Komponenten definiert. Ferner wird die `complex`-Anweisung benötigt, um im Sinne der Konfiguration (siehe Kapitel 9) gebundene Konfigurationen anzugeben.

Für die `complex`-Anweisungen wird für jede neu definierte Komponente `<PATH>` vom Komponenten-Framework `<TYPE>` ein Pict-Tupel mit dem gleichen Namen `<PATH>` erzeugt. Dazu wird der Pict-Prozessagent `new_<TYPE>` mit den angegebenen Parameter-Komponenten (`<PC1>, ...,<PCn)>`) aufgerufen, der dieses Tupel `<PATH>` zurückgibt. Dieses Tupel repräsentiert dann die Komponente zur Laufzeit.

```
transComplexComp(complex <PATH> <TYPE> (<PC1>, ..., <PCn>) )
     :=
     val <PATH> = (new_<TYPE> <PC1>,..., <PCn>)
```

7.3.11 Erzeugung des Empfangskanals: `transProcCalls`

Eine Prozedur (und auch eine Methode) wird – wie in Pict üblich – durch einen Kanal repräsentiert, auf dem die Argumente einer Prozedur sowie ein Return-Kanal für eventuell berechnete Rückgabewerte zur Bearbeitung übergeben werden. Bei der Übersetzung von SCDL müssen dabei zwei Aspekte berücksichtigt werden: Zum einen muss eine Prozedur überhaupt aufgerufen werden können. Zum anderen darf eine Prozedur nur dann ausgeführt werden, wenn es die Verhaltensbeschreibung der Komponente oder Klasse zulässt. Aus diesem Grund werden die beiden Aspekte „Aufruf“ und „Ausführung“ voneinander getrennt, indem mittels der Übersetzung durch `transProcCall` pro SCDL-Prozedur `<P>` ein Kanal `<P>` definiert wird, auf dem ein Aufruf angenommen wird. Mittels der Übersetzung von `transProcedure` (siehe 7.3.12) wird dann pro Prozedur `<P>` ein Prozessagent `proc_<P>` definiert, über den die Funktionalität der Prozedur dann letztlich aufgerufen werden kann.

Die Signatur des Kanals einer Prozedur ist die gleiche wie in der Type-Definition der Komponente bzw. Klasse (`transChannelType`).

```
transProcCall( procedure <P>(<FPARAS>) = ... )
     :=
     new <P> :^[ transChannelType(<FPARAS>) ]
```

7.3.12 Übersetzung einer Prozedur: `transProcedure`

Der Rumpf einer Prozedur wird in einen Prozessagenten übersetzt, der als formale Parameter alle `IN`- und `INOUT`-Parameter und einen zusätzlichen Return-Kanal besitzt, in dem die veränderten bzw. neuen `INOUT`- bzw. `OUT`-Parameter zurückgegeben werden.

Es sei `<ID>(<D`$_1$`> <T`$_1$`> <P`$_1$`>, ..., <D`$_n$`> <T`$_n$`> <P`$_n$`>) = <BODY>` die Definition einer Prozedur/Methode und es gilt:

$$
\begin{aligned}
& m \in \{1,\ldots,n\} \\
\wedge \quad & o \in \mathbb{N} \\
\wedge \quad & i_1,\ldots,i_m,i_{m+1},\ldots,i_o \in \{1,\ldots,n\}
\end{aligned}
$$

```
∧    i1<...<im
∧    im+1<...<io
∧    Di1, ..., Dim ∈ {in, inout}
∧    Dim+1, ..., Dio ∈ {inout, out}
```

Dann ist die Übersetzung einer Prozedur/Methode wie folgt definiert:

```
transProcedure( <ID>(<D1> <T1> <P1>, ..., <Dn> <Tn> <Pn>) = <BODY>)
        :=
    def proc_<ID> [<Pi1>:<Ti1> <Pi2>:<Ti2>...<Pim>:<Tim>
                              return:^[<Tim+1> ... <Tio>]]
        = transProcBody(<BODY>)
```

7.3.13 Übersetzung des Prozedurrumpfes: `transProcBody`

Die Übersetzung eines Prozedurrumpfes kann einfach durch ein sequentielles Übersetzten aller Aktionen des Rumpfes – z.B. durch eine rekursive Abarbeitung – realisiert werden. Hier werden deswegen nur die Regeln für die Übersetzung der einzelnen Aktionen in die Sprache Pict vorgestellt, aber nicht die sequentielle Abarbeitung, die sich daraus leicht konstruieren lässt.

Leseaktion (RECV-Aktion):

Wartet eine Prozedur auf den Empfang einer Nachricht bzw. den Aufruf einer Prozedur `<ID>`, so wird dies durch eine Pict-Leseaktion auf dem Kanal `<ID>` abgebildet. Wird auf diesem Kanal ein Tupel geschickt, so wird daraufhin die zugehörige Prozedur `proc_<ID>` (siehe Abschnitt 7.3.12) aufgerufen und die berechneten Ergebniswerte über den Return-Kanal zurückgegeben.

Es seien $\langle D_1\rangle\ \langle T_1\rangle\ \langle P'_1\rangle, \ldots, \langle D_n\rangle\ \langle T_n\rangle\ \langle P'_n\rangle$ die formalen Parameter der Prozedur `<ID>`, und es gilt

```
        die aktuellen Parameter P1,...,Pn sind zu den Typen T1,...,Tn kompatibel,
∧    m ∈ {1,...,n}
∧    o ∈ |N
∧    i1,...,im,im+1,...,io ∈ {1,...,n}
∧    i1<...<im
∧    im+1<...<io
∧    Di1, ..., Dim ∈ {in, inout}
∧    Dim+1, ..., Dio ∈ {inout, out}
```

Dann ist die Übersetzung einer RECV-Aktion folgendermaßen definiert:

```
transProcBody( <ID>?( <P1>, ..., <Pn>) )
    :=
    <ID>?[<Pi1> ... <Pim> return] =
        (new return2:^[<Tim+1> ... <Tio>]
        (proc_<ID>![<Pi1> ... <Pim> return2] |
        return2?[<Pim+1> ... <Pio>] =
```

```
        (return![<Pi_{m+1}> ... <Pi_o>] | ... /*Rest des Processrumpfs*/
        )))
```

Schreibaktion (SEND-Aktion):

Ein Aufruf einer Prozedur <PATH> wird in Pict durch eine Schreibaktion auf den Kanal <PATH> abgebildet. Zum Aufruf wird neben den zu übergebenden Parametern auch ein Return-Kanal benötigt, auf dem die Ergebnisse zurückgeliefert werden. Es seien $<D_1>$ $<T_1>$ $<P'_1>$, ..., $<D_n>$ $<T_n>$ $<P'_n>$ die formalen Parameter der Prozedur <PATH>, und es gilt

die aktuellen Parameter $P_1, \ldots, P_n$ sind zu den Typen $T_1, \ldots, T_n$ kompatibel,

$\wedge$ $m \in \{1, \ldots, n\}$

$\wedge$ $o \in \mathbb{N}$

$\wedge$ $i_1, \ldots, i_m, i_{m+1}, \ldots, i_o \in \{1, \ldots, n\}$

$\wedge$ $i_1 < \ldots < i_m$

$\wedge$ $i_{m+1} < \ldots < i_o$

$\wedge$ $Di_1, \ldots, Di_m \in \{in, inout\}$

$\wedge$ $Di_{m+1}, \ldots, Di_o \in \{inout, out\}$

Dann ist die Übersetzung einer SEND-Aktion folgendermaßen definiert:

```
transProcBody( <PATH>!( <P_1>, ..., <P_n>) )
     :=
     (new return:^[<Ti_{m+1}> ... <Ti_o>]
     (<PATH>![<Pi_1> ... <Pi_m> return] |
     return?[<Pi_{m+1}> ... <Pi_o>] = ... /*Rest des Processrumpfs*/
     ))
```

Parallele Komposition:

Die parallele Komposition kann direkt nach Pict übersetzt werden.

```
transProcBody( <ProcessBody1> | <ProcessBody2> )
     :=
     transProcBody(<ProcessBody1>) | transProcBody(<ProcessBody2>)
```

Erzeugen neuer Objekte:

Objekte können innerhalb einer Komponente neu erzeugt werden. Ein Objekt der Klasse <T> wird durch Aufruf des New-Operators erzeugt und an die Variable <P> gebunden. In Pict wird dazu der Prozessagent new_<T> der Klasse <T> aufgerufen, der einen Kanal liefert, auf dem ein Tupel mit den Methoden dieses Objektes geschickt wird.

```
transProcBody( new <T>:<P> )
     :=
     (new ref_<P>:^<T>
     (new_<T>![ref_<P>] |
     ref_<P>?<P> = ... /* Rest des Prozessrumpfs */
```

```
))
```

Abschluss einer Prozedur durch "return":

Nachdem eine Prozedur abgearbeitet wurde, müssen die Ergebnisse an die `INOUT`- und `OUT`-Parameter übergeben werden. Dazu werden sie auf den zum Anfang der Prozedur angelegten Return-Kanal geschrieben.

```
transProcBody( return(<P1>,...,<Pn>) )
     :=
     return![<P1> ... <Pn>]
```

7.3.14 Übersetzung des Verhaltens einer Komponente/Klasse: `transBehaviour`

Die Verhaltensdefinitionen von Komponenten und Klassen setzen sich aus einer Reihe von Zustands-/Transitionsbeschreibungen zusammen. In SCDL werden Zustände als Situationen verstanden, in denen eine Komponente auf eine Interaktion mit einer anderen Komponente (= Eintreten eines Ereignisses, d.h. Aufruf einer Prozedur) wartet. Transitionen sind Übergänge zwischen Zuständen. Tritt in einem Zustand ein Ereignis ein, so werden innerhalb der Transition eine Reihe von Aktionen (= Aufrufe von Prozeduren/Methoden von (anderen) Objekten) durchgeführt und dann in einen Folgezustand gewechselt. Diese Schematisierung des Verhaltens ist vergleichbar mit der von objektorientierten Statecharts (siehe [Harel und Gery 1997]), durch die jedoch ausschließlich das Verhalten von Objekten, aber nicht von Komponenten beschrieben werden kann und für die bislang keine anerkannte formale Definition des Verhaltens existiert.

Zustands-/Transitionsbeschreibungen werden in Pict als lokale Prozessagenten abgebildet. Zustände werden als Signaturen von Prozessagenten und Transitionen als Körper des Prozessagenten repräsentiert. Der erste Zustand einer Komponente bzw. einer Klasse ist der Default-Zustand `initialized`.

Zum Ende einer Transition wird entweder ein neuer Prozessagent oder der leere Prozess aufgerufen. Ein neuer Prozessagent verkörpert dann einen Folgezustand. Eine Zustands-/Transitionsbeschreibung kann durch Verwendung eines Auswahloperators (Guarded Choice, siehe Abschnitt 17.2.7) auch mehrere alternative Transitionen beinhalten.

Die Übersetzung der Transitionen (siehe Abschnitt 7.3.15) erfolgt genauso wie die Übersetzung der Prozeduren, allerdings mit dem Unterschied, dass den Zuständen ausschließlich `IN`-Parameter übergeben werden dürfen, d.h. ein Zustand kann keine Informationen zurückgeben.

```
transBehaviour(<ID>(<P1> <T1>, ..., <Pn> <Tn>) = <BODY>)
     :=
     def <ID> [<P1>:<T1> ... <Pn>:<Tn>]=
```

```
transTransition(<BODY>)
```

7.3.15 Übersetzung einer Transition: `transTransition`

Eine Transition kann ähnlich wie der Rumpf einer Prozedur SEND- und RECV-Aktionen enthalten. Ebenfalls können mit dem new-Operator neue Objekte erzeugt werden. Transitionen können parallel zusammengesetzt werden (Parallele Komposition). Die Übersetzung dieser Konstrukte folgt dem gleichen Schema wie in Abschnitt 7.3.13 und wird deswegen hier nicht noch einmal angegeben.

Zusätzlich kann jedoch eine Transition durch ein Guarded-Choice-Konstrukt (siehe Abschnitte 17.2.6 und 17.2.7) erweitert werden, wodurch zum Beispiel alternative Zustandswechsel realisiert werden können. Zudem ist der Zustandswechsel selbst eine Besonderheit der Beschreibung einer Transition.

Auswahloperator:

Ein Auswahloperator kann durch ein Guarded Choice realisiert werden, der allerdings im Gegensatz zum klassischen π-Kalkül in Pict nicht direkt formuliert werden kann und deswegen durch den im Teil V in Abschnitt 17.2.7 vorgestellten Mechanismus realisiert wird. Zwei ECA-Regeln werden dann in das folgende Schema eingebunden.

```
transTransition( <ECARULE1> or <ECARULE2> )
     :=
     sync![(event
               transECARule( <ECARULE1> )
               transECARule( <ECARULE2> )
          )]
```

Übersetzung von ECA-Regeln:

ECA-Regeln bestehen aus einem Ereignis (Nachricht oder Prozeduraufruf), einer Bedingung und einer Aktion. Die Aktion – in SCDL wird sie als Transition bezeichnet – wird nur dann ausgeführt, wenn das Ereignis eingetreten und die Bedingung erfüllt ist. In Abschnitt 17.2.7 wird hierzu ein Pict-Prozessagent `eventCondAction` entwickelt, der direkt genutzt werden kann.

Die Bedingung erhält als aktuelle Parameter eine Teilmenge der Parameter des Ereignisses. Diese Parameter dürfen nur `IN`-Parameter sein, da die Bedingung den Zustand einer Komponente oder eines Objektes nicht verändern darf.

Es seien $\langle D_1 \rangle$ $\langle T_1 \rangle$ $\langle P'_1 \rangle$, ..., $\langle D_n \rangle$ $\langle T_n \rangle$ $\langle P'_n \rangle$ die formalen Parameter der SCDL-Prozedur `<IDE>` (= Ereignis bzw. Prozedur), und es gilt,

die aktuellen Parameter $P_1, \ldots, P_n$ sind zu den Typen $T_1, \ldots, T_n$ kompatibel,

```
∧    m ∈ {1,...,n}
∧    o ∈ |N
∧    i1,...,im,im+1,...,io ∈ {1,...,n}
∧    i1<...<im
∧    im+1<...<io
∧    Di1, ..., Dim ∈ {in, inout}
∧    Dim+1, ..., Dio ∈ {inout, out}
```

Ferner seien (in <S1> <Q´1>, ..., in <Sk> <Q´k>, out Bool b) die formalen Parameter der Prozedur <IDC> (= Bedingung), und es gilt

```
     die aktuellen Parameter Q1,...,Qk sind jeweils vom Typ S1,...,Sk ,
∧    j1, ...,jk ∈ {1,...,n}
∧    Q1, ...,Qk ∈ {P1,...,Pn}
```

Dann ist die Übersetzung einer ECA-Regel folgendermaßen definiert:

```
transECARule( on <IDE>?(<P1>,...,<Pn>)
              if <IDC>!(<Q1>,...,<Qk>)
              do <PROCBODY> )
    :=
    (eventCondAction #[<Ti1> ... <Tim> ^[<Tim+1> ... <Tin>]]
         <IDE>
         \[[<Pi1>:<Ti1> ... <Pim>:<Tim>] b:^Bool] = <IDC>![<Pj1> ... <Pjk> b]
         \[<Pi1>:<Ti1> ... <Pim>:<Tim> return:[<Tim+1> ... <Tin>]] =
                   (new return1:^[<Tim+1> ... <Tin>]
                   (proc_<IDE>![<Pi1> ... <Pim> return1] |
                   return1?[<Pim+1> ... <Pin>] =
                   (return![<Pim+1> ... <Pin>]  |
                   transTransition( <PROCBODY> )
                   )))
    )
```

Zustandswechsel

Wird ein neuer Prozessagent aufgerufen, d.h. eine Transition innerhalb der Verhaltensbeschreibungen wurde beendet und der nächste Zustand wird erreicht, dann wird dieser Wechsel als Aufruf eines anderen Prozessagenten repräsentiert.

Sei <ID>(<T1> <P´1>, ..., <Tn> <P´n>) die Signatur des neuen Zustands. Dann wird der zugehörige Aufruf <ID>!(<P1>,...,<Pn>) folgendermaßen übersetzt:

```
transTransition( <ID>!(<P1>,...,<Pn>) )
    :=
    <ID>![<P1> ... <Pn>]
```

7.3.16 Default-Semantik

Um den Aufwand für die Erstellung von Komponentenbeschreibungen zu verringern, gibt es zwei Arten von Beschreibungen, die weggelassen werden dürfen und für die dann

Standardbeschreibungen (Default-Semantiken) angenommen werden. Dies sind die Beschreibungen von Prozedur- bzw. Methodenrümpfen sowie die Verhaltensbeschreibungen von Komponenten/Klassen.

7.3.17 Standardbeschreibung für Prozeduren/Methoden

Wird der Rumpf einer Prozedur/Methode weggelassen, wird angenommen, dass alle `INOUT`-Parameter unverändert zurückgegeben werden und für alle `OUT`-Parameter ein neues Objekt erzeugt wird. So wird z.B. die folgende Definition der Prozedur `test` in der Übersetzung automatisch um eine Beschreibung ergänzt, die ein neues Objekt des Typs `Z` erzeugt und es durch den Parameter `z` referenziert zurückgibt.

```
test(in x X, inout y Y, out z Z);
     ==>
test(in x X, inout y Y, out z Z) = new z‘:Z; return(y,z‘)
```

7.3.18 Standardverhaltensbeschreibung für Komponenten/Klassen

Die Verhaltensbeschreibung einer Komponente/Klasse ist optional. Wird kein Zustand mit dem Namen `initialized` definiert, so wird ein Standardverhalten angenommen, in dem jede Prozedur/Methode in jeder Situation aufgerufen werden darf. Diese Beschreibung lässt sich durch eine Auswahl auf Basis des Guarded Choice realisieren, die ausschließlich aus ECA-Regeln besteht, in denen die Ereignisse wiederum aus den Prozedur- bzw. Methodensignaturen bestehen, die Bedingungen (Condition) immer erfüllt (`true`) sind und die Aktionen (Resttransitionen) nur den Wechsel in den Ausgangszustand (`initialized`) beschreiben.

Hat z.B. eine Komponente zwei Prozeduren mit Signaturen s_1 und s_2, so wird das Verhalten folgendermaßen vorausgesetzt:

```
initialized = on s₁ if true do initialized
         or   on s₂ if true do initialized
```

D.h. nach einem Aufruf der Prozedur mit Signatur s_i wird sofort wieder in den initialen Zustand gewechselt, so dass wieder beide Prozeduren aufgerufen werden können.

7.4 Formale Traces

Wie schon in Abschnitt 3.3.2 angedeutet wurde, spielen Traces bzw. die aus ihnen gewonnenen Geschäftsprozesse in dieser Arbeit eine besondere Rolle.

Traces eines Prozesskalküls sind endliche Sequenzen von Kommunikationen (Interaktionen) zwischen Prozessen (siehe z.B. [Olderog 1991]). Traces in Pict basieren auf Interaktionen, die durch das Kommunikationsverhalten auf Basis der Reduktionsrelation (siehe Abschnitt

17.2.3) definiert sind. SCDL-Traces entstehen nun auf der Grundlage der Interaktionen zwischen Komponenten, die durch die Übersetzung von Methoden- und Prozeduraufrufen nach Pict definiert sind.

Komponentenbeschreibungen, die in SCDL formuliert werden, sind ausführbare Programme. Durch die Übersetzung der Komponenten in Prozesse des asynchronen π-Kalküls können die Programme in einer Laufzeitumgebung für Pict (siehe [Pierce und Turner 1997]) direkt ausgeführt werden. Traces lassen sich somit aus protokollierten Programmabläufen in Pict gewinnen.

Ein *SCDL-Trace* ist eine endliche Folge von SCDL-Interaktionen. Eine *SCDL-Interaktion* ist als Aufruf einer Methode/Prozedur definiert, der durch die SEND-Aktion einer Komponente (siehe Abschnitt 7.3.13) eingeleitet, dann durch eine weitere Komponente in einer RECV-Aktion abgearbeitet und letztendlich durch die Rückgabe einer (evtl. leeren) Liste von `INOUT`- und `OUT`-Parametern abgeschlossen wird.

SCDL-Interaktionen können relativ einfach automatisch protokolliert werden. Dabei ist es sinnvoll, Aufrufe von Prozeduren (inkl. Methoden) und Rückgaben von Parametern als Klammern zu verstehen, die andere Interaktionen, die auch wieder als Klammern aufgebaut sind, umschließen. Wie in Programmiersprachen üblich, können Interaktionsbeginn und -ende z.B. mit den Schlüsselworten CALL und EXIT gekennzeichnet werden.

In SCDL geschriebene Systeme können hochgradig nebenläufig sein. Insofern gilt es bei der Protokollierung der Interaktionen nicht nur die interagierenden Objekte, sondern auch die Prozesse innerhalb der Objekte zu beachten. Alle Komponente und alle erzeugten Objekt stellen eigenständige Prozesse dar. Innerhalb eines solchen Prozesses können weitere Prozesse durch parallele Komposition (siehe Abschnitt 7.3.12) initiiert werden.

In dieser Arbeit werden Interaktionen als PROLOG-Tupel protokolliert, wodurch die automatische Weiterverarbeitung im Rahmen der Implementierung (siehe Kapitel 11) vereinfacht wird. Ein Aufruf (`call`) einer Prozedur bzw. Methode hat folgende Struktur:

```
call(PID, CALLER, ID, PROCEDURE, ACTUAL_PS, CALLEE, ID2, FORMAL_PS)
```

`PID`	die Nummer des Prozesses, aus dem ein Aufruf erfolgt,
`CALLER`	der Typ der aufrufenden Komponente bzw. des aufrufenden Objekts,
`ID`	die aufrufende Komponente bzw. das aufrufende Objekt (Instanz),
`PROCEDURE`	der Name der aufgerufenden Methode/Prozedur,
`ACTUAL_PS`	die Liste der übergebenen Parameter (als Tupel: DIR/TYPE/NAME/ID), DIR = Richtung des Parameters, TYPE= Typ des Parameters, NAME=Name des Parameters und ID = das tatsächlich übergebene Objekt,
`CALLEE`	der Typ der aufgerufenen Komponente bzw. des aufgerufenen Objekts,
`ID2`	die aufgerufene Komponente bzw. das aufgerufene Objekt und
`FORMAL_PS`	die Liste der formalen Parameter der Prozedur.

Die Rückgabe (`exit`) der `INOUT`- und `OUT`-Parameter wird durch den Return-Befehl (siehe Abschnitt 7.2) eingeleitet und durch Tupel der folgenden Form protokolliert:

```
exit(PID, ID, RETURN_PS)
```

`PID`	die Nummer des Prozesses, aus dem der Aufruf ursprünglich erfolgte,
`ID`	die aufgerufene Komponente bzw. das aufgerufene Objekt und
`RETURN_PS`	die Liste der zurückgegebenen Parameter (als Tupel: DIR/TYPE/NAME/ID), DIR = Richtung des Parameters, TYPE= Typ des Parameters, NAME=Name des Parameters und ID = das übergebene Objekt.

Transitionen in SCDL haben immer auch einen Zustandswechsel[1] (siehe Abschnitt 7.3.15) zur Folge – sowohl auf der Sender- als auch auf der Empfängerseite. Zur Veranschaulichung von Folgen von Interaktionen lassen sich diese Zustandsinformationen zusätzlich nutzen. Sie können unmittelbar nach einer Interaktion aufgeführt werden. Zustände werden folgendermaßen protokolliert:

```
state(PID, TYPE, ID, STATE, ACTUAL_PS)
```

`PID`	die Nummer des Prozesses, aus dem ein Aufruf erfolgte,
`TYPE`	der Typ der Komponente, die ihren Zustand wechselt,
`ID`	die Komponente, die ihren Zustand wechselt,
`STATE`	der Name des neuen Zustands und
`ACTUAL_PS`	Referenzen auf andere Objekte, die für den Zustand des Objekts relevant sind.

In Abbildung 7.4 wird ein Trace dargestellt, der auf der SCDL-Beschreibung des Beispiels aus Abschnitt 7.1 basiert. In diesem Beispiel werden vier Komponenten (`server`, `client`, `csFramework`, `asp`) definiert, von denen zwei (`server` und `client`) je ein Objekt (`o5` vom Typ `client.Client` und `o4` vom Typ `server.Service`) erzeugen. Danach wird die Methode `use` des Objekts `o5` mit dem Objekt `o4` als Parameter aufgerufen, die wiederum die Methode `serve` des Objekts `o4` aufruft. Alle Objekte und Komponenten haben nur einen Zustand (`initialized`), in den sie nach Ausführung einer Prozedur zurückkehren.

Der dargestellt Trace wurde mit Hilfe der im Rahmen dieser Arbeit erstellten Implementierung (siehe Teil III) automatisch generiert.

[1] Im einfachsten Fall wechselt ein Objekt oder eine Komponente immer wieder in den gleichen Zustand.

```
    state(p1,server,o1,initialized, []),
    state(p2,client,o2,initialized, []),
    state(p3,asp,o3,initialized, []),
call(p3,asp,o3, createService, [], server,o1, []),
exit(p3,o1, server, [(s,server.Service,o4),  []]),
    state(p4,server.Service,o4,initialized, []),
    state(p3,server,o1,initialized, []),
call(p3,asp,o3, createClient, [], client,o2, []),
exit(p3,o2, client, [(x_2,client.Client,o5),  []]),
    state(p5,client.Client,o5,initialized, []),
    state(p3,client,o2,initialized, []),
call(p3,asp,o3, use, [(in,sv,serverC.Service,o4)],
        client.Client,o5, [(in,s,serverC.Service,o4)]),
    call(p3,client.Client,o5, serve, [], server.Service,o4, []),
    exit(p3,o4, server.Service, [ []]),
exit(p3,o5, client.Client, [ []]),
    state(p3,server.Service,o4,initialized, []),
    state(p3,client.Client,o5,initialized, []).
```

Abbildung 7.4: Trace zum Beispiel

8 Normsprachen und Komponenten

Eine Komponente wird im Rahmen ihrer Entwicklung durch verschiedene Dokumente wie z.B. CASE-Tool-Dokumente in UML, technische Handbücher oder Benutzungshandbücher beschrieben. Da Komponenten allgemein einsetzbar sein sollen, können diese Beschreibungen nicht von vornherein auf einen konkreten Anwendungsbereich abzielen, denn Sinn einer wiederverwendbaren Komponente ist es ja gerade, von möglichst vielen Anwendungsbereichen zu abstrahieren. So müsste es sinnvollerweise für jeden Anwendungsbereich ein eigenes Handbuch geben. Das gleiche gilt somit auch für SCDL-Beschreibungen.

Grundsätzlich kann man sagen, dass es umso wichtiger wird, die Komponenten auf Grundlage der jeweiligen Begriffswelt des Anwendungsbereiches zu erklären, je stärker man sich im Gestaltungsprozess des Anwendungssystems (vgl. Abschnitt 1.2 und Abbildung 8.1) dem Anwender nähert. Die Forderung nach zunehmender Konkretisierung gilt folglich auch für Anwendungsarchitekten und Konfigurierer. Während der Entwicklung der Komponenten wird möglichst stark von potentiellen Anwendungsbereichen abstrahiert. Durch die Komposition von Komponenten und durch den konkreten Einsatz beim Anwender nähert man sich jedoch wieder einem konkreten Anwendungsbereich. Dazu erscheint es sinnvoll, Begriffe einer Branche, eines Betriebstyps und eines konkreten Unternehmens als *Interpretationen* in die Beschreibung einer Komponente einzubringen, damit später schneller und besser ein Verständnis für eine Komponente und ein ganzes Anwendungssystem aufgebaut werden kann.

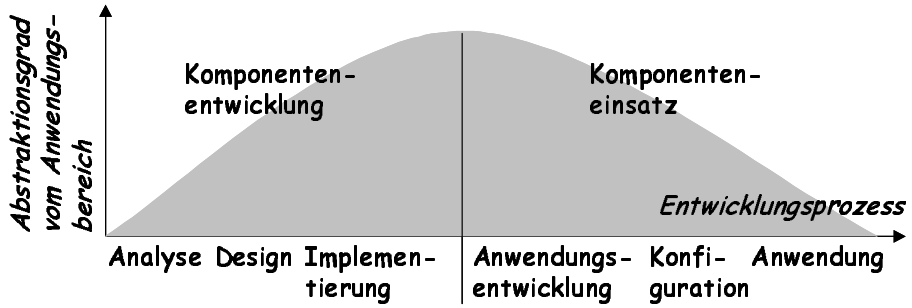


Abbildung 8.1: Abstraktionsgrade im Gestaltungsprozess

Durch die zunehmende Abstraktion der Softwareentwicklung werden auch die Beschreibungen der Komponenten zunehmend abstrakter und zielen weniger auf die inhaltlichen, sondern mehr auf die logischen Zusammenhänge ab. Ein Problem dabei ist, dass natürlichsprachliche Sätze in einem Handbuch direkt verstanden werden können, Sätze einer abstrakten, symbolischen Sprache, die ausschließlich „die Logik“ einer Komponente hervorheben, jedoch einer zusätzlichen Interpretation ihrer Symbole bedürfen – zumindest aus Sicht der Anwender.

SCDL-Beschreibungen, wie sie in Kapitel 7 erarbeitet wurden, sind in diesem Sinne formale Beschreibungen, die ebenfalls nur die „Logik“ (Verhaltenslogik) einer Komponente beschreiben, aber nicht auf bestimmte Begriffe eines Anwendungsbereiches verweisen. So ist es aus dieser formalen Sicht beispielsweise gleichgültig, ob eine Klasse „Bestellung“ oder „b001“ heißt. Nur im Zusammenhang komponentenübergreifender Vereinbarungen (Standards), z.B. auf Basis von Kontrakten, spielen Symbolvereinbarungen eine Rolle.

Um einem Leser einer SCDL-Beschreibung die inhaltliche Interpretation einer Komponente explizit zu ermöglichen, werden in diesem Kapitel Abbildungen eingeführt, mit denen er SCDL-Beschreibungen durch *Annotationen* ihrer Symbole und Aussagenformen (Signaturen) ergänzen kann. Für den Anwender wären dazu u.a. natürlichsprachliche Interpretationen sinnvoll, die jedoch die Probleme der natürlichen Sprache (siehe Abschnitt 5.1) mit sich bringen würden. Aus diesem Grund werden Worte und Sätze in der in Kapitel 5 vorgestellten Normsprache formuliert. Die Ideen, die den folgenden Abschnitten zugrunde liegen, wurden in der Arbeit von Sandmann (siehe [Sandmann 1998]) am Beispiel der Statecharts untersucht.

8.1 Normsprachliche Kommunikation zwischen Komponenten

Kommunikation zwischen zwei Partnern – oder wie in dieser Arbeit: zwischen Komponenten sowie zwischen Komponenten und Benutzern – basiert i.d.R. auf unterschiedlichen *Äußerungstypen* (siehe Abschnitt 5.5 und [Ortner 1997]). Grundsätzlich kann man mindestens die Typen *Behauptungen* (*Aussagen*), *Aufforderungen* und *Fragen* unterscheiden, die auf die Satzarten aus Abschnitt 5.5 zurückgeführt werden können. In Normsprachen werden diese Äußerungstypen durch unterschiedliche Satzarten gekennzeichnet.

Die Analogie zwischen der menschlichen und der Kommunikation von Softwarekomponenten drückt sich in der Informatik durch die metaphorische Verwendung der Begriffe der menschlichen Kommunikation aus. So gibt es beispielsweise Kommunikationsmodelle, die auf einer einfachen Frage-Antwort-Beziehung (Request/Reply-Beziehung, siehe Abschnitt 3.3.4) basieren. In diesen Modellen können einige Komponenten – beispielsweise Anwendungskomponenten – „Fragen stellen“ und andere Komponente – beispielsweise zentrale Datenbankkomponenten – „Antworten (Aussagen) liefern“. In anderen Kommunikationsmodellen (Publish/Subscribe) schicken spezielle Komponenten nach entsprechender Aufforderung Nachrichten („Aussagen“), die von den anfordernden Komponenten verarbeitet oder auch ignoriert werden können.

Nun stellt sich die Frage, wie man diese Formen der Kommunikation normsprachlich ausdrücken kann. Da bislang nur auf Aufforderungen und Aussagen eingegangen wurde, soll nun ein Vorschlag zur Formulierung von Fragen gemacht werden, der auf Ideen zurück geht, die aus der Logik stammen und auch in Logikprogrammiersprachen wie PROLOG (siehe [Clocksin und Mellish 1987]) Verwendung finden. In PROLOG werden Fragen als *Aussageformen* verstanden, die aus Aussagen durch das Austauschen von Subjekten durch

Variablen erzeugt werden. Diese verallgemeinerten – *variablen* – Aussagen können durch einen Inferenzmechanismus mit einer Wissensbasis abgeglichen werden. Eine Frage kann beantwortet werden, wenn es „Erfüllungen“ (Spezialisierungen) der Aussageform gibt.

Ein Beispiel: Folgende Aussagen seien gegeben[1]:

```
[“Herbert”]   ε [Vater]  [von-, “Gustav”].
[“Liese”]     ε [Mutter] [von-, “Gustav”].
[“Herbert”]   ε [Vater]  [von-, “Erwin”].
```

Möchte man nun wissen, von wem Herbert der Vater ist, würde man die Frage folgendermaßen formulieren:

```
[“Herbert”] ε [Vater] [von-, X].
```

Diese Frage kann man nun mit den gegebenen Aussagen vergleichen. Man erkennt, dass die Variable X alternativ durch zwei Individuen substituiert werden kann. Die Aussageform und eine Aussage können also durch Substitution angeglichen („gematcht“) werden. In PROLOG wird in diesem Zusammenhang von *Unifikation* gesprochen. Somit kann die Frage beantwortet werden: Herbert ist Vater von Gustav und Erwin.

Ähnlich wie bei der schon bekannten Aufforderung, die zum Herbeiführen eines Zustandes oder einer Handlung auffordert, kann man nun eine Frage auch als Aufforderung zur Erfüllung einer Aussageform verstehen. Dazu wird die Aussageform in die Aufforderung aufgenommen:

```
[“Jürgen”] ! [erfülle] [[“Herbert”] ε [Vater] [von-, X]].
```

Umgangssprachlich könnte diese Aufforderung in „Jürgen, sag mir, von wem Herbert der Vater ist!“ übersetzt werden. In der Normsprache kann man dann folgende Kurzform vereinbaren:

```
Frage    ::= Nominator “?” Aussageform
```

8.1.1 Aussageformen

Ungeklärt ist bisher, welche Aussageformen zugelassen werden sollen. In den angeführten Beispielen wurden bisher nur Individuen durch Variablen ersetzt. Denkbar ist, dass alle Wörter – mit Ausnahme der Kopulae – durch Variablen ersetzt werden. Jedoch scheint es von Vorteil zu sein, wenn man die Prädikatoren einer Aussage trotzdem beibehalten kann. Dadurch werden gezielte Fragen ermöglicht. So würde die Frage „Von welcher Tochter ist

[1] Auf die Einführung der PROLOG-Schreibweise wird hier verzichtet. Statt dessen wird die in Kapitel 5 definierte Normsprache genutzt.

Herbert der Vater?“ verlangen, dass man der Variablen, die in einer Aussageform das Wort „welchen“ darstellt, den Prädikator „Frau“ zusprechen würde. Normsprachlich könnte die Frage dann folgende Form haben:

```
X ε Frau.
! [“Herbert” ε [Vater] [von-, X ]]
```

Oder in einer abgekürzten Schreibweise:

```
! [“Herbert” ε [Vater] [von-, (X : Frau)]]
```

Es wird hier also eine Typisierung der Variablen vorgenommen, wie sie aus verschiedenen Programmiersprachen her bekannt ist.

Als gültige Aussageformen werden nun für jeden Prädikatoren-Subterm p der Grammatik aus Abschnitt 5.8 auch die Teilaussageform (X : p) zugelassen. Individuen können durch die gleichen Teilaussageformen ersetzt werden. Da die Grammatik hier nicht nochmals komplett aufgeführt werden soll, werden nur die Grammatiksymbole aufgezählt, die auf die vorgestellte Art und Weise ersetzt werden können. Dies sind:

```
Subjekt, GPrädikation, Obj1, Obj2, Obj3 und alle Terminale und Nicht-
Terminale, die von diesen abgeleitet werden.
```

Die Einführung von Variablen kann also sowohl für Eigenprädikatoren als auch für Geschehnisprädikatoren und Präpositionen usw. vorgenommen werden.

8.1.2 Unifikation

Unter der Unifikation wird der Abgleich einer Aussageform mit einer Aussage verstanden, um zu prüfen, ob sich die Aussageform durch die Aussage erfüllen lässt. Bei der Unifikation stellen sich folgende Fragen:

1. Wird nur ein rein struktureller Abgleich zugelassen oder können auch strukturell, d.h. syntaktisch, unterschiedliche Sätze unifiziert werden?

2. Dürfen nur Aussageformen mit Aussagen (a) unifiziert werden oder können auch Aussagen mit Aussagen (b) und Aussageformen mit Aussageformen (c) unifiziert werden? Welche Bedeutung haben diese Arten der Unifikation?

zu 1.) Es soll zugelassen werden, dass Aussagen auch kürzere Aussageformen erfüllen dürfen. „Kürzer“ bedeutet, dass die Aussageformen weniger Angaben zu den Subjekten und Objekten erfordern, als sie die Aussagen tatsächlich liefern. Die Fragen verlangen also allgemeinere Antworten, als eine Aussage sie liefert. Da zusätzliche Informationen den Fragenden natürlich nicht „stören“, kann diese Art der Unifikation ruhig zugelassen werden.

zu 2.) Es werden nur die beiden ersten Arten der Unifikation (a und b) benötigt: Die Unifikation einer Aussageform, die keine Variablen enthält und somit eigentlich wieder eine Aussage ist (b), mit einer „richtigen“ Aussage, kann als Frage interpretiert werden, die mit „ja“ beantwortet werden kann, wenn die erste Aussage Teil der zweiten Aussage ist oder beide Aussagen identisch sind. Im anderen Fall würde die Frage mit „nein“ beantwortet werden.

8.1.3 Formalisierung der Unifikation

In diesen Abschnitt wird der Vorgang der Unifikation formalisiert. Es werden folgende Kombinationen der Unifikation zugelassen: Aussageformen mit Aussagen und Aussagen mit Aussagen. Da beide Fälle Spezialfälle der Unifikation zweier Aussageformen sind, kann man sich auf diese Kombination beschränken. Dazu wird zunächst der Vorgang der Substitution, also der Ersetzung der Variablen durch konkrete Prädikatoren und Nominatoren, definiert.

Definition (in Anlehnung an [Schöning 1987])

Sei A eine Aussageform, X eine Variable und t eine Prädikation oder ein Nominator. Dann bezeichnet A[x/t] diejenige Aussage(-form), die man erhält, indem jedes Vorkommen der Variable x in A durch den Term t ersetzt wird. Durch [x/t] wird eine *Substitution* beschrieben. Substitutionen können auch zu Folgen von Substitutionen der Form $[x_1/t_1]...[x_n/t_n]$ erweitert werden. Eine solche Substitution wird dann als schrittweises Anwenden der Teilsubstitutionen verstanden.

Definition

Eine Substitution *sub* ist ein *Unifikator* einer endlichen Menge von Aussageformen $A = \{A_1, ... A_n\}$, g.d.w. A_1 *sub* = A_2 *sub* ... = A_n *sub*. D.h., durch Anwenden der Substitution *sub* auf jede Aussageform in A entsteht ein und dieselbe Aussage(-form). Man sagt dann auch „A ist unifizierbar“.

Ein Unifikator *sub* einer Menge von Aussagen A heißt *allgemeinster Unifikator* von A, falls für jeden Unifikator *sub‘* von A gilt, dass es eine Substitution sub_2 gibt mit *sub‘* = *sub* sub_2.

Der Satz von Robinson beweist, dass jede Menge von unifizierbaren Aussageformen auch einen allgemeinsten Unifikator besitzt (siehe dazu [Schöning 1987]).

8.2 Interpretation der Komponentenbeschreibungen

Ein kleines, aber typisches Beispiel für wiederverwendbare Komponenten ist in den vergleichbaren, aber zueinander komplementären Anwendungsbereichen Ein- und Verkauf in einem Unternehmen zu finden. In beiden Bereichen hat man es mit externen Organisationen zu tun: im Einkauf mit Lieferanten und im Verkauf mit Kunden. Es hat sich in verschiedenen Branchen als zweckmäßig herausgestellt, eine allgemeine Komponente zu entwickeln, mit der

sowohl Lieferanten als auch Kunden verwaltet werden können. Begrifflich wird dann in einer solchen Komponente abstrahiert, indem man die Gemeinsamkeiten von Lieferanten und Kunden herausstellt und diese z.B. unter dem Begriff „Geschäftspartner“ zusammenfasst. Es ist zu betonen, dass der Begriff „Geschäftspartner“ methodisch nicht vor den Begriffen „Lieferant“ und „Kunde“ konstruiert wurde, sondern Ergebnis einer zweckgebundenen Analyse der Gemeinsamkeiten dieser beiden Begriffe ist – im Sinne einer ökonomischen Softwareentwicklung.

Existiert nun eine Komponente zur Verwaltung von Geschäftspartnern, die zudem in einen Framework für integrierte betriebliche Anwendungssysteme sowohl für den Einkauf als auch für den Verkauf einsetzbar ist – sie entspricht also zwei entsprechenden Kontrakten –, dann sollte die vorhergehende Abstraktion für eine konkrete Anwendung wieder rückgängig gemacht werden können. D.h., dass die Komponente im Kontext eines Anwendungsbereiches *interpretiert* werden soll. Denn für das Verständnis des Anwenders, aber auch des Anwendungsarchitekten und des Konfigurierers (zu den verschiedenen Rollen siehe Abschnitt 1.2) ist es förderlich, dass je nach Anwendungsbereich im Einkauf von „Lieferanten“ und im Verkauf von „Kunden“ und nicht allgemein von „Geschäftspartnern“ die Rede ist.

Aus dieser Tatsache wird unmittelbar ersichtlich, dass sich dem abstrahierenden Entwicklungsprozess einer Komponente auch wieder ein entsprechender spezialisierender Prozess (Interpretation) bei der Konfiguration eines Anwendungssystems anschließen muss (siehe Abbildung 8.1). Dabei sind verschiedene Spezialisierungsdimensionen erkennbar. Zum einen kann eine Komponente entsprechend den Begrifflichkeiten einer Branche/Teilbranche spezialisiert werden. Da aber selbst innerhalb eines Unternehmens unterschiedliche Spezialisierungen für eine Komponente existieren können – man betrachte zum Beweis nur das kleine Kunden-Lieferanten-Beispiel –, muss es zusätzlich gemäß der Funktionsbereiche eines Unternehmens mehrere Konkretisierungen geben dürfen.

Konkretisierungen werden dadurch erreicht, dass die Symbole und Satzformen der Komponenten wieder eine Bedeutung erlangen. Dies soll mittels normsprachlicher Annotationen geschehen, die man einer Komponentenbeschreibung anhängt. Mindestens zwei Arten von Annotationen kann man somit festmachen:

1. Wie angedeutet werden im Rahmen der Softwareentwicklung die ursprünglichen Begriffe der Anwendungsbereiche zu inhaltlich (fast) leeren Symbolen abstrahiert. Im Einsatzfall einer Komponente können nun die Symbole wieder mit Prädikatoren des (evtl. neuen) Anwendungsbereiches belegt werden, die für den Anwender eine Bedeutung haben – also wieder Begriffe sind.

2. Wie in Abschnitt 5.5 dargestellt wurde, gibt es in natürlichen Sprachen wie auch in der dargestellten Normsprache verschiedene Satzarten, die allgemein zur Beschreibung von Geschehnissen, Handlungen, Fähigkeiten usw. herangezogen werden können. Merkmal

der auf die abstrakte Logik abzielenden Softwareentwicklung (siehe z.B. [Wedekind 1992]) ist es, diese Informationen aus ihren Satzformen zwecks Abstraktion zu entfernen, da sie aus Sicht der maschinellen Verarbeitung unwichtig sind. Wie in der Prädikatenlogik üblich werden auch in der Softwareentwicklung Sachverhalte nur noch in tupel-ähnlichen Satzformen dargestellt. Für die Annotationen einer Komponente kann es jedoch hilfreich sein, auf quasi natürlichsprachliche (normsprachliche) Satzformen zurückzugreifen, aus denen z.B. sofort erkennbar ist, ob eine Aussage aus dem Anwendungsbereich einen Zustand oder eine Handlung beschreibt. Deshalb sollte man nun die formalen Sätze wieder mit „Leben füllen“ können, in dem man Elementarsätzen wieder eine konkretere Form gibt.

8.2.1 Normsprachliche Annotationen

In diesem Abschnitt wird nun ein Vorschlag unterbreitet, wie man die Konkretisierung systematisch im Rahmen der Konfiguration einer Komponente vornehmen kann.

Dazu können die Bestandteile (Prozeduren, Methoden, Bezeichner, ...) einer Komponente mittels Abbildungsfunktionen umbenannt werden. Dabei wird nicht nur auf einzelne Bezeichner abgezielt, sondern auch auf komplexere Ausdrücke wie Aussagen und Anfragen einer Komponente, die durch die Abstraktion der Softwareentwicklung für den Anwender oft bis zur Unkenntlichkeit „verstümmelt“ wurden. Die Bezeichner können systematisch konkretisiert werden, so dass der Anwender die Komponente wieder interpretieren und „verstehen“ kann.

Dabei ist natürlich zu beachten, dass es ja gerade Ziel der Abstraktion war, möglichst viele Anwendungsbereiche abzudecken. Andersherum muss es nun bei der Konkretisierung einer Komponente möglich sein, möglichst viele Konkretisierungen zuzulassen. Das bedeutet u.a. auch, dass neue Konkretisierungen angestrebt werden können, an die bei der Entwicklung einer Komponente noch gar nicht gedacht wurde. Wie Beispiele aus der Praxis verdeutlichen, ist dieser Sachverhalt keineswegs unwahrscheinlich. So wurde z.B. in [Möhle 1997] gezeigt, wie u.a. eine Projektplanungskomponente auch zur Produktionsplanung eingesetzt werden kann, obwohl bei der ursprünglichen Entwicklung der Komponente nicht an diese Anwendungsmöglichkeit gedacht wurde.

In dieser Arbeit werden nun insbesondere zwei Arten der Konkretisierung ermöglicht: Symbolannotationen und Annotationen von Ausdrücken (Aktions- und Zustandsbeschreibungen).

Unter Symbolannotation soll die Möglichkeit verstanden werden, Symbole der Komponentenbeschreibungen, die zur Laufzeit Entitäten des Anwendungsbereiches klassifizieren, durch spezielle Begriffe dieses Anwendungsbereiches zu ersetzen. Die Entitäten des Anwendungsbereiches werden typischerweise durch Komponenten und Klassen vorgegeben.

Ausdrücke, z.B. in Prozedursignaturen und Zustandsbeschreibungen, können durch normsprachliche Satzformen annotiert werden.

8.2.2 Annotationen von Komponenten und Klassen

Die Symbole, die in der SCDL-Beschreibung einer Komponente eingeführt und genutzt werden, basieren hauptsächlich auf den Namen der Komponenten, Kontrakte, Klassen und Objekttypen (siehe Abschnitt 7.2) sowie ihrer Zustände und Prozeduren (inkl. Methoden). Der Spezialfall der Kontrakte (inklusive Objekttypen) wird hier vernachlässigt. Kontrakte bzw. Objekttypen können genauso behandelt werden wie Komponenten bzw. Klassen.

Wie aus der SCDL-Syntaxdefinition ersichtlich wird, kann eine Komponente eindeutig über einen Pfadnamen (<PATH>) identifiziert werden. Die Klassen $<ID_1>$, ..., $<ID_n>$ einer Komponente <PATH> lassen sich genauso referenzieren, indem dem Klassenname der Pfad der Komponente, in der die Klasse definiert ist, vorangestellt wird (<PATH>.$<ID_i>$, $i \in \{1,..,n\}$).

Die Prädikatoren (siehe Abschnitt 5.4) und ihre Bedeutung, also die Begriffe einer Anwendungswelt, müssen systematisch in einem Lexikon (siehe Abschnitte 5.1 und 5.9) geführt werden. Um Komponenten- und Klassennamen nun durch die Begriffe eines Anwendungsbereiches eines Unternehmens konkretisieren zu können, wird pro Komponente eine Abbildung K definiert, die der Komponente und jeder Klasse einen Prädikator aus diesem Lexikon zuordnet, der für den jeweiligen Begriff steht. Die Menge der Prädikatoren der Begriffswelt des Anwendungsbereiches wird mit Π bezeichnet. Die (partielle) Abbildung hat dann folgende Signatur[1]:

$$K: L(<PATH>) \rightarrow \Pi$$

Die Menge aller Abbildungen dieses Typs wird mit $\mathfrak{K}$ bezeichnet.

8.2.3 Beispiel

Auf Basis des Beispiels aus Abschnitt 7.1 werden die vorgestellten Abbildungen nun verdeutlicht. Die Komponente `client` benötigt einen Dienst (`Service`), den sie einfach nur „benutzt“. Eine Interpretation für eine solche Komponente könnte beispielsweise ein Internet-Benutzer sein, der als Kunde (`client.Client`) eines Online-Shops einen

---

[1] Bezeichner in spitzen Klammern („<“, „>“) dienen in Abschnitt 7.2 der Definition von Nichtterminalen in der SCDL-Grammatik. Gemäß der Vereinbarungen in Abschnitt 16.2 wird die Menge von Worten, die ausgehend von einem Nichtterminal <X> mittels einer Grammatik abgeleitet werden kann, mit L(<X>) bezeichnet.

virtuellen Warenkorb (`server.Service`) benötigt. Die entsprechende Abbildungen – hier der Einfachheit halber als Menge von Tupeln definiert – könnten dann folgendermaßen aussehen:

K_{client} = {(client, Kundenverwaltung), (client.Client, Kunde)}

Entsprechend könnten die Komponente `server` als Warenkorbverwaltung, die virtuelle Warenkörbe zur Verfügung stellt, und die Komponente `asp` als ein alle anderen Komponenten umfassender Online-Shop interpretiert werden:

K_{server} = {(server, Warenkorbverwaltung), (server.Service, Warenkorb)}
K_{asp} = {(asp, Online-Shop)}

8.2.4 Annotationen von Aktionen

Die Interaktion zweier Komponenten verläuft über das Wechselspiel von SEND- und RECV-Aktionen (siehe Abschnitt 7.3.12). Erst wenn sich zwei „passende“ Aktionen „gefunden“ haben, ergibt sich eine Interaktion. Da die Funktionalität einer Interaktion auf Seiten des Empfängers beschrieben wird, wird auch dort die Annotation einer Interaktion zugeordnet. Die Menge der möglichen RECV-Aktionen (Prozeduren) einer Komponente werden durch die Signaturen der direkten und indirekten Prozeduren (Methoden) definiert. Eine Prozedur kann über die Konkatenation des Pfadnamens der Komponente bzw. der Klasse und ihres eigenen Namens eindeutig referenziert werden.

Die Bedeutung einer Interaktion (siehe Abschnitt 8.1) kann vielfältig sein. So kann eine Interaktion u.a.

- als eine Anfrage (SEND-Aktion) an eine Service-Komponente verstanden werden, die durch die RECV-Aktion eine entsprechende Antwort liefert (Beispiel: Eine Komponente schickt eine SQL-Anfrage an eine Datenbankkomponente und erhält eine Antwort in Form einer Menge von Tupeln),

- eine Interaktion i.S. einer Kooperation sein, an der sowohl die sendende als auch die empfangende Komponente beteiligt ist (Beispiel: Zwei Softwareagenten erarbeiten zusammen eine Lösung),

- eine Aufforderung einer Komponente an eine andere sein, etwas durchzuführen (Beispiel: Eine Komponente fordert eine Druckkomponente auf, etwas zu drucken) oder

- aus einer einfachen Mitteilung/Behauptung (Aussage) bestehen.

Alle diese Sachverhalte lassen sich durch Sätze der in Kapitel 5 vorgestellten Normsprache beschreiben. Den Signaturen der Prozeduren $<P_1>,...,<P_n>$ einer Komponente oder Klasse <PATH> können nun über die Abbildung A normsprachliche Sätze mit Variablen

(Aussageformen, siehe Abschnitt 8.1.1) zugeordnet werden. Eine Signatur einer Prozedur <P> aus einer Klasse oder Komponente <PATH> kann dann durch <PATH>.<P> eindeutig referenziert werden. Diese Referenz kann durch die EBNF-Produktion ausgedrückt werden:

```
<SIG>          ::= <PATH>“.”<ID>
```

Die partielle Abbildung A aus der Menge der Prozeduren und Methoden kann dann wie folgt definiert werden:

A: L(<SIG>) → {s | s ist eine normsprachliche Aussageform. Als Variablen kommen ausschließlich Variablen der Signatur und `caller` sowie `callee` vor}

Die formalen Parameter der Prozedur können als Platzhalter in der normsprachlichen Aussageform genutzt werden. Zwei besondere Platzhalter sind die Variablen `caller` und `callee`, die die aufrufende bzw. aufgerufene Komponenteninstanz oder ein Objekt einer Klasse repräsentieren. Durch das Einsetzen der Platzhalter entstehen also Aussageformen, die im Rahmen der konkreten Interaktion zwischen zwei Komponenten, bei der die formalen Parameter durch aktuelle Parameter ersetzt werden, zu konkreten Aussagen unifiziert (siehe Abschnitt 8.1.2) werden.

Die Mengen aller Abbildungen dieses Typs werden mit $\mathcal{A}$ bezeichnet.

8.2.5 Beispiel (Fortsetzung)

Die Annotationen zu den Prozeduren des Beispiels werden folgendermaßen definiert.

```
A_client = {
     (client.createClient(out x:client.Client),
               “caller π identifizieren x [mit- callee]”),
     (client.Client.use(in x:server.Service),
               “callee!callee π benutzen x”),
}
A_server = {
     (server.createService(out s:server.Service),
               “caller π zur_Verfügung_stellen s [mit- callee]”),
     (server.Service.serve(),
               “caller π benutzen <callee>”)
     }
A_asp = {}
```

Die Prozedur `createClient` ermöglicht der aufrufenden Komponente mittels der Kundenverwaltung einen Kunden zu identifizieren. Mit der Prozedur `use` wird ein Kunde (`client.Client`) aufgefordert, einen Warenkorb (`server.Service`) zu benutzen.

Die Prozedur `createClient` ermöglicht der aufrufenden Komponente mittels der Einkaufskorbverwaltung einen Warenkorb zur Verfügung zu stellen. Mit der Prozedur `serve` kann der Warenkorb benutzt werden.

Die Komponente `asp` besitzt keine Prozeduren. Folglich ist die Abbildung leer.

8.2.6 Annotationen von Zuständen

Jede Komponente und jede Klasse kann eine Verhaltensspezifikation beinhalten, die mit dem initialen Zustand `initialized` beginnt. Ein Zustand wird durch ein Symbol, das in Bezug auf die Komponente bzw. Klasse eindeutig ist, bezeichnet. Zusätzlich können dem Zustand eine Liste von Parametern mitgegeben werden, auf den sich spätere Aktionen beziehen können. Die Parameter sind ausschließlich `IN`-Parameter, da ein Zustand nicht wie eine Prozedur Informationen zurückgeben kann. Ferner ist der Zustand lokal zu einer Komponente bzw. Klasse, d.h. es gibt im Vergleich zu den Annotationen keine aufrufende oder aufgerufene Komponente bzw. Klasse, sondern immer nur die Komponente (Klasse) selbst. Diese Komponente wird durch die Konstante `this` referenziert. Mit Ausnahme der aufgezählten Unterschiede kann die Annotation von Zuständen genauso definiert werden, wie die von Aktionen (siehe Abschnitt 8.2.4).

Die partielle Abbildung Z von der Menge der Zustände, die durch den Pfad (<PATH>) ihrer Komponente bzw. Klasse und dem Namen des Zustands (<STATE>), referenziert werden, kann dann wie folgt definiert werden:

Z: L(<PATH>“.”<STATE>) → {s | s ist eine normsprachliche Aussageform.
Als Variablen kommen ausschließlich Variablen aus <STATE> und this vor}

Die Mengen aller Abbildungen dieses Typs werden mit $\mathcal{Z}$ bezeichnet.

8.2.7 Beispiel (Fortsetzung)

Im Beispiel in Abschnitt 7.1 haben alle Komponenten und Klassen nur einen Zustand (`initialized`) zu dem sie immer wieder zurückkehren. Dennoch können die Zustände der Komponenten und Klassen im Anwendungsbereiche „Online-Shop“ unterschiedlich interpretiert werden.

```
Z = {(asp.initialized, “this ε bereit”),
     (client.initialized, “this ε bereit”),
     (client.Client.initialized, “this ε identifizieren”),
     (server.initialized, “this ε bereit”),
     (server.Service.initialized, “this ε zur_Verfügung_stehen”)
}
```

Der Online-Shop, die Kunden- und die Warenkorbverwaltung sind alle, nachdem sie zur Laufzeit erzeugt wurden, gestartet und „bereit“. Ein Kunde der Kundenverwaltung ist in dem Zustand „identifiziert“ und der Warenkorb „steht“ nach seiner Erzeugung „zur Verfügung“.

Die formale Trace-Beschreibung aus Abschnitt 7.4 kann, wenn man die Annotationen durch die Abbildungen K, A und Z berücksichtigt, normsprachlich wie in Abbildung 8.2 dargestellt werden. Offensichtlich können die mit Annotationen versehenden Komponenten in dem jeweiligen Kontext (hier: Anwendungsbereich „Online-Shop“) leichter durch den Anwender verstanden („interpretiert“) werden als die rein formalen Bezeichner.

state(p1,server,o1,initialized, [])
(o1:Warenkorbverwaltung) ε bereit.
state(p2,client,o2,initialized, [])
(o2:Kundenverwaltung) ε bereit.
state(p3,asp,o3,initialized, [])
(o3:Online-Shop) ε bereit.
call(p3,asp,o3, createService, [], server,o1, [])
(o3:Online-Shop) π zur_Verfügung_stellen (o4:Warenkorb) [mit- (o1:Warenkorbverwaltung)].
exit(p3,o1, server, [(s,server.Service,o4), []])
state(p4,server.Service,o4,initialized, [])
(o4:Warenkorb) ε zur_Verfügung_stehen.
state(p3,server,o1,initialized, [])
(o1:Warenkorbverwaltung) ε bereit.
call(p3,asp,o3, createClient, [], client,o2, [])
(o3:Online-Shop) π identifizieren (o5:Kunde) [mit- (o2:Kundenverwaltung)].
exit(p3,o2, client, [(x_2,client.Client,o5), []])
state(p5,client.Client,o5,initialized, [])
(o5:Kunde) ε identifizieren.
state(p3,client,o2,initialized, [])
(o2:Kundenverwaltung) ε bereit.
call(p3,asp,o3, use, [(in,sv,serverC.Service,o4)],
client.Client,o5, [(in,s,serverC.Service,o4)])
(o5:Kunde)! (o5:Kunde) π benutzen (o4:Warenkorb).
call(p3,client.Client,o5, serve, [], server.Service,o4, [])
(o5:Kunde) π benutzen (o4:Warenkorb)
exit(p3,o4, server.Service, [[]])
exit(p3,o5, client.Client, [[]])
state(p3,server.Service,o4,initialized, [])
(o4:Warenkorb) ε zur_Verfügung_stehen.
state(p3,client.Client,o5,initialized, [])
(o5:Kunde) ε identifizieren.

Abbildung 8.2: Trace mit Zuordnung von Annotationen

8.2.8 Konsistenz der Annotationen

Durch die SCDL-Komponentenbeschreibungen werden verschiedene Beziehungen (Subordination, Komposition, Parametrierung) zwischen den Symbolen definiert, die auch in den Abbildungen K, A und Z berücksichtigt werden können. So werden z.B. durch die Parametrierung zwischen Komponenten und Kontrakten, Subordinationsbeziehungen definiert, die aus technischer Sicht der Substituierbarkeit (siehe Abschnitt 4.1.2) dienen. Aus inhaltlicher Sicht können diese Beziehungen als Unter-Oberbegriffbeziehungen interpretiert werden. Sollen z.B. einem SCDL-Objekttyp `Order` und einer zu diesem Objekttyp kompatiblen SCDL-Klasse `SalesOrder` Prädikatoren zugeordnet werden, so sollten diese

im Sinne des Anwenders auch eine Unter-Oberbegriffbeziehung haben (z.B. „Auftrag“ und als Spezialisierung „Kundenauftrag“). Diese Beziehungen werden in Normsprachen typischerweise in einem *Lexikon* (siehe Abschnitte 5.1 und 5.9) verwaltet. Ein solches Lexikon kann auf die SCDL-Beschreibungen zurückgreifen und entsprechende Beziehungen – neben den Unter-Oberbegriffbeziehungen z.B. auch Teile-Ganzes-Beziehungen – automatisch überprüfen bzw. propagieren. Auf diese Idee wird im Weiteren nicht mehr eingegangen.

8.3 Annotationen für Komponenten

Die bislang definierten Annotationen wurden Teilen der Komponentenbeschreibungen direkt zugeordnet. Jedoch sind einfache Beispiele konstruierbar, wo dies nicht ausreicht. Nimmt man wie in Abschnitt 8.2 an, dass eine Komponente zur Verwaltung von Geschäftspartnern sowohl für den Einkauf (Lieferantenverwaltung) als auch für den Vertrieb (Kundenverwaltung) geeignet ist, so sollten die Annotationen in Abhängigkeit des Anwendungsbereiches, also zweimal vorgenommen werden können.

Aus diesem Grund sollen Annotationen sich fortan auf ihren Anwendungsbereich beziehen dürfen. Dazu können die Abbildungen aus $\mathcal{A}$, $\mathcal{K}$ und $\mathcal{Z}$ wie bislang direkt (Fall 1) oder indirekt über die Slots von Framework-Komponenten (Fall 2) einer Komponente zugeteilt werden

Im ersten Fall werden Abbildungen K, A und Z einer konkreten Komponente <PATH> durch die Abbildung I zugeteilt.

$$I : L(\langle PATH \rangle) \rightarrow \mathcal{K} \times \mathcal{A} \times \mathcal{Z}$$

Im zweiten Fall (Parametrierung) werden die beiden Abbildungen den Slots zugeordnet, die innerhalb einer Komponente <PATH> durch einen Namen <ID> identifiziert werden können.

$$S: L(\langle PATH \rangle \text{“.”} \langle ID \rangle) \rightarrow \mathcal{K} \times \mathcal{A} \times \mathcal{Z}$$

Natürlich sind noch detailliertere Zuordnungen denkbar. So kommt es in Anwendungsbereichen durchaus vor, dass, die Prädikatoren für Objekte einer Klasse von den Attributbelegungen oder allgemein vom Zustand der Objekte abhängig sind. Die „Rolle“, von der ein Objekt abhängig ist, kann also zur Laufzeit wechseln. Beispielsweise kann ein Formular, das anfangs als „Auftrag“ diente, durch einen Vermerk „ausgeliefert“ zu einer Rechnung werden. Entsprechende Mechanismen werden hier nicht betrachtet. Siehe zum Thema „Role Modeling“ z.B. [Kappel, Retschitzegger und Schwinger 1998].

9 Anwendungssoftwareprodukte

Die Bestrebungen der Komponentenorientierung zielen u.a. darauf ab, Anwendungssysteme grundsätzlich flexibel zu gestalten und es dem Anwender zu ermöglichen, bestehende Komponenten schnell und einfach durch bessere/modernere Komponenten zu ersetzen. Zu beachten ist jedoch, dass es mit steigender Verfügbarkeit alternativer Komponenten auf dem Softwaremarkt zunehmend komplizierter wird, die Vielfalt möglicher Konfigurationen zu überschauen und zu „managen". Werden heute betriebliche Anwendungssoftwareprodukte en bloc dem Kunden angeboten, so besteht in Zukunft zumindest die Möglichkeit diese aus Komponenten verschiedener Hersteller zu kombinieren und dabei möglichst auf die besten („best of breed") zurückzugreifen.

Neben der Auswahl müssen die jeweils resultierenden Konsequenzen einer möglichen Auswahl bewertet und die für den Anwender beste Konfiguration[1] ausgewählt werden. Da der Anwender diesen komplexen Prozess kaum alleine bewerkstelligen kann, wird in dieser Arbeit die Auffassung (siehe Hypothese 3 in Abschnitt 1.4) vertreten, dass Spezialisten, sogenannte Anwendungsarchitekten, diese Aufgabe übernehmen, indem sie für ausgewählte Anwenderkreise wie z.B. Branchen mögliche Konfigurationsalternativen vorbereiten, aus denen Unternehmen die für sie beste Konfiguration auswählen können. Die Anwendungsarchitekten übernehmen auch die Garantie[2] – u.U. unter Rückversicherung bei den Komponentenherstellern – für die reibungslose Zusammenarbeit (Kompatibilität, Substituierbarkeit, siehe Abschnitt 4.1.2) der ausgewählten Komponenten. Eine so vorbereitete Menge von Konfigurationsalternativen soll *Anwendungssoftwareprodukt* heißen. Anwendungssoftwareprodukte sind somit nicht sofort einsetzbar, sondern müssen vor dem Einsatz durch einen Konfigurierer an die Anforderungen des Anwenders angepasst (konfiguriert) werden.

In diesem Kapitel wird nun ein Vorschlag unterbreitet, wie die Konfigurationsalternativen eines Anwendungssoftwareprodukts beschrieben werden können. Dazu wird eine weitere Sprache, die *Component Configuration Language* (CCL), definiert. Auf Basis dieser Sprache kann eine Menge von Konfigurationsalternativen zu einer Menge von Komponenten

---

[1] Die Bewertung einer Konfiguration ist sicherlich kein triviales Problem. Unterschiedliche Faktoren, wie funktionaler Abdeckungsgrad, Prozessorientierung und Integrationsgrad der Funktionen sowie eine angestrebte Kostenersparnis durch eine Konfiguration müssen quantitativ bewertet werden (können).

[2] Unter „Garantie" wird hier der juristische Aspekt dieses Wortes verstanden. Eine 100%ig Zusicherung der technisch einwandfreien Zusammenarbeit von Komponenten gibt es in der Praxis bislang nicht.

angegeben werden. Einige Ideen zu diesem Kapitel wurden im Rahmen einer Diplomarbeit (siehe [Jaekel 1999]) näher untersucht, in der u.a. ein System zur Konfiguration des Klassen-Frameworks *San Francisco* von IBM (siehe [Abinava et al. 1998]) entwickelt wurde.

9.1 Begriffe der Konfiguration

Ein einziges Produkt, insbesondere ein betriebliches Softwareprodukt, kann i.d.R. nicht gleichzeitig die verschiedenen Anforderungen aller Kunden (Unternehmen) erfüllen. Um besser auf die individuellen Bedürfnisse eingehen zu können, werden heute oft nicht mehr nur einzelne Produkte, sondern ganze Produktfamilien (engl. product lines) entworfen. Die verschiedenen Produkte einer Produktfamilie bestehen größtenteils aus identischen Produktteilen und einer kleineren Menge von Produktteilen, die sich von Produkt zu Produkt unterscheiden. So kann man beispielsweise einen Modelltyp eines Autoherstellers (z.B. VW Golf) als Produktfamilie auffassen, für den die meisten Produktteile identisch sind, jedoch verschiedene Motorausstattungen, Fahrwerke usw. als alternative Produktteile zur Auswahl angeboten werden. In einem *Konfigurationsprozess* (siehe Abschnitt 9.2.4) werden dann in mehreren Konfigurationsschritten entsprechend der Anforderungen des Kunden und unter Berücksichtigung erlaubter Teilekombinationen – so verlangt z.B. eine höhere Motorisierung auch meist ein entsprechendes Fahrwerk oder zumindest bessere Reifen – nach und nach die verschiedenen alternativen Komponenten ausgewählt und zu einem kundenspezifischen System zusammengesetzt.

Die *Konfigurationsaufgabe* kann durch zwei wesentliche Eigenschaften charakterisiert werden:

1. Das zu konfigurierende System setzt sich aus *Komponenteninstanzen* einer vorgegebenen Menge von wohldefinierten *Komponenten(typen)* zusammen.

2. Die Komponenteninstanzen[1] arbeiten in einer vorgegebenen Weise zusammen.

Etwas genauer bedeutet das, dass die *Konfiguration* (siehe [Bach 1994]) als Aufgabe verstanden werden kann, bei der

- eine feste Menge vordefinierter Komponenten,
- eine Menge von *Kompatibilitätsinformationen*, die angeben, wie Komponenten verbunden werden dürfen,

[1] Im Weiteren wird der Einfachheit halber meist von Komponenten statt von Komponenteninstanzen gesprochen. Nur da, wo es zur Unterscheidung notwendig ist, wird explizit von Komponenteninstanzen die Rede sein.

- eine Menge von anwendungsspezifischen *Konfigurationsplänen*, die Prozeduren zur Auswahl und Komposition von Komponenten definieren, und
- eine Menge von *Anforderungen* durch den späteren Nutzer/Anwender eines Systems

vorgegeben werden, und zu deren Zielerreichung entweder

- eine Anwendung der Konfigurationspläne auf Grundlage der vorgegebenen Komponenten gefunden werden muss, die alle Kompatibilitätsrestriktionen und Anforderungen des Nutzers/Anwenders einhält, oder
- gezeigt werden muss, dass es eine solche Anwendung, die die Restriktionen und Anforderungen einhält, nicht gibt.

Diese Definition unterscheidet drei Arten von Wissen: Zuerst das Wissen über die Komponenten und darüber, wie die Komponenten zusammengesetzt werden dürfen (*Anwendungswissen*), zweitens das Wissen darüber, wie eine Konfiguration erzeugt wird (*Kontrollwissen*), und drittens das Wissen darüber, wie diese Wissensquellen angewandt werden können. Das Anwendungswissen (Kompatibilitätsinformationen) entspricht dem Wissen über Substituierbarkeit verschiedener Softwarekomponenten und zusätzlicher Konsistenzbedingungen (Constraints). Das Kontrollwissen (Konfigurationspläne) wird im nächsten Kapitel unter der Überschrift „Geschäftsprozessorientierte Konfiguration“ näher behandelt.

Wie schon weiter oben angedeutet wurde, heißt eine Kombination von Komponenten, die für einen bestimmten Zweck, z. B. für eine bestimmte Umgebung oder einen Benutzer, ausgewählt worden ist, eine *Konfiguration*. Man kann gebundene, generische und abstrakte Konfigurationen unterscheiden. *Gebundene Konfigurationen* sind eindeutige Konfigurationen (z. B. Produktfreigaben), die unabhängig vom jeweiligen Kontext wie Zeit oder dem Zustand anderer Komponenten sind. Ein *UND/ODER-Graph* ohne ODER-Knoten (vgl. auch 3.4.3), indem also für jede Auswahlmöglichkeit eine Entscheidung getroffen wurde, stellt beispielsweise eine gebundene Konfiguration dar. Eine *generische Konfiguration* ist ebenfalls eindeutig beschrieben, jedoch abhängig von einem Kontext. Beispielsweise ist eine Konfiguration, die von jeder benötigten Komponente die neueste Version wählt, generisch.

Durch unvollständige Beschreibungen entstehen abstrakte Konfigurationen. *Abstrakte Konfigurationen* beschreiben Mengen von Konfigurationen und müssen weiter zu gebundenen Konfigurationen verfeinert werden.

9.2 Konfiguration von Softwarekomponenten

9.2.1 Konfiguration von Anwendungssystemen

Die Konfiguration eines Anwendungssystems findet zu verschiedenen Zeitpunkten des Softwareentwicklungsprozesses statt. Der klassische Forschungsbereich des „Software Configuration Management (SCM)“ (siehe z.B. [Berlack 1992]) hat sich in der Vergangenheit vornehmlich mit der Konfiguration beim und durch den Softwarehersteller befasst. Ursprünglich waren Softwareprodukte für den Kunden relativ inflexibel, aber dafür sofort einsetzbar. Die verschiedenen funktionalen Varianten, die durch unterschiedliche Anforderungen der Anwender (Unternehmen) zustande kamen, wurden beim Hersteller verwaltet und zum Bau großer Systeme genutzt. Um jedoch dem Kunden mehr Flexibilität bieten zu können, verlagerte sich der Prozess der Konfiguration zunehmend zum Anwender. So wurden die Konfigurationswerkzeuge, die ursprünglich dem Entwickler zum Verwalten der Komponenten und zum Erstellen eines konkreten Softwaresystems dienten, dem Anwender an die Hand gegeben (siehe [Hufgard 1994]), damit er selbstständig entscheiden konnte, welche Funktionen einer Softwareproduktfamilie genutzt werden sollten und welche nicht. Diese Tatsache befreite den Softwarehersteller jedoch nicht von dem Management verschiedener (zeitlicher) Versionen und von Varianten, die z.B. aufgrund des Entwicklungsfortschritts bzw. unterschiedlicher Betriebs- und Datenbankmanagementsysteme immer noch benötigt wurden.

Heute besteht während des gesamten Softwarelebenszyklus der Bedarf, die Bestandteile eines Produkts bzw. einer Produktfamilie zu verwalten und gegebenenfalls neu zu konfigurieren. Softwarekonfigurationsmanagement beinhaltet die Organisation und Verwaltung von Softwareelementen (Modulen, Komponenten, Bibliotheken usw.) von der Entwicklung über die erste Konfiguration bis hin zur kontinuierlichen Verbesserung von Anwendungssystemen. Die verschiedenen Verfahren der Konfiguration unterscheiden sich je nach Phase und Softwareprodukt stark voneinander. So setzen heutige monolithische Anwendungssoftwareprodukte wie das R/3-System auf datenbankbasierte Ansätze, wie sie schon in Abschnitt 2.1.4 vorgestellt wurden.

9.2.2 Grundlegende Anforderungen an Konfigurationsmodelle

Um eine Konfiguration so flexibel wie möglich zu gestalten, müssen einige grundlegende Anforderungen an ein Modell eines Softwarekonfigurationsmanagements (*SCM-Modell*) erfüllt sein, die relativ unabhängig vom eigentlichen Konfigurationsproblem sind.

In [Zeller 1997] werden folgende drei Grundanforderungen gestellt:

Verwaltung von Mengen: Nur wenige SCM-Modelle ermöglichen die Spezifikation von Variantenmengen und folglich von Konfigurationsmengen (= abstrakte Konfigurationen). Variantenmengen sollen solange durch weitere Informationen verfeinert werden können, bis eine Konfiguration gebunden ist. Sie sorgen ferner dafür, dass mehrere Konfigurationen parallel verarbeitet werden können und dadurch Alternativen nicht frühzeitig ausgeschlossen werden. Ein SCM-Modell sollte also mengenorientiert und nicht elementorientiert sein.

Attributierung: Die Komponenten eines SCM-Modells sollten attributierbar sein. Darüber lassen sich Eigenschaften, die für den Konfigurationsprozess wichtig sind, und die im Normalfall implizit in der Komponente verborgen sind, in das Systemmodell mit aufnehmen. Attribute dienen der Identifikation von Komponenten, der Propagierung von Informationen für Konsistenzbedingungen (Constraints) und der Darstellung von Beziehungen zwischen Komponenten.

Attributausdrücke (Propagierung): Um Abhängigkeiten (Konsistenzbedingungen, configuration constraints) zwischen den Komponenten definieren zu können, bedarf es einer Möglichkeit, Bedingungen zwischen den Attributen der Komponenten zu formulieren und Ergebnisse zwischen den Komponenten zu propagieren. Die einfachste Möglichkeit besteht darin, boolesche Ausdrücke zu formulieren, die durch die Attributbelegungen der Komponenten erfüllt werden müssen.

Da im besonderen Maße logikbasierte Sprachen die geforderten Aspekte berücksichtigen, werden in [Zeller 1997] deswegen die potentiellen Kandidaten „Logik erster Stufe“, „Beschreibungslogiken“ und „Feature-Logiken“ miteinander verglichen. Die Ergebnisse können folgendermaßen zusammengefasst werden:

Logik erster Stufe ist zu generell. Alle geforderten Eigenschaften wie Verwaltung von Mengen, Attributierung und Propagierung lassen sich grundsätzlich zwar ausdrücken, jedoch müssen entsprechende Mechanismen dazu explizit als Axiome und Regeln erst noch formuliert werden.

Beschreibungslogiken stellen explizit Konstrukte zur Beschreibung von Mengen (concept descriptions) und zur Attributierung (Roles) zur Verfügung. Die Konzeptbeschreibungen dienen gleichzeitig der Beschreibung von Konsistenzbedingungen. Als einziger Nachteil wird in [Zeller 1997] angeführt, dass die Propagierung von Werten der Attribute einer Komponente zu ihren übergeordneten Komponenten (Komposita) explizit formuliert werden muss.

Feature-Logiken stellen eine Teilmenge der Beschreibungslogiken dar. Sie sind insofern eingeschränkt, als dass die Attribute (Features) im Gegensatz zu den Attributen der Beschreibungslogiken funktional sind: Jedes Attribut hat genau einen Wert. Die Attributwerte eines Kompositums (= komplexe Komponente) werden implizit über die Attribute der Komponenten ermittelt. Der Nachteil der Feature-Logiken ist, dass nur Konjunktionen von

Attribut-Wert-Paaren formuliert werden können. Negationen und Disjunktionen, wie sie in vielen Beschreibungslogiken ansonsten möglich sind, werden nicht erlaubt.

Prinzipiell ließen sich auch *Constraint-Satisfaction-Ansätze* (siehe z.B. [Van Hentenryck 1989]) für die Konfigurationsproblematik nutzen. Sie sind aber nicht mengenorientiert, da sie i.d.R. nach genau einer (möglichst optimalen) Lösung streben. Da es sich bei der Konfiguration um einen hochgradig interaktiven Prozess handelt, für den das Optimum zudem kaum zu formalisieren ist, braucht dieser Ansatz nicht weiter betrachtet werden.

Entscheidender Nachteil der Feature-Logiken scheint die Tatsache zu sein, dass Features funktional sind. Denn es sind einfache Problemfälle zu konstruieren, in denen eine beliebige Anzahl, also Mengen von Komponenten (siehe „parallele Erweiterung“, Abschnitt 3.2.2), ausgewählt werden dürfen. Dazu wäre allerdings auch eine Erweiterung im Komponentenmodell (siehe Kapitel 7) notwendig, damit Slots auch tatsächlich mehrere Komponenten aufnehmen können.

Für diese Arbeit haben sich Beschreibungslogiken als durchaus brauchbar herausgestellt. So wurde in der Arbeit von Jaekel (siehe [Jaekel 1999]) am Beispiel San Francisco gezeigt, wie sich ein Konfigurationssystem für Klassen-Frameworks aufbauen lässt. Beschreibungslogiken sollen deshalb auch als Basis für das hier zu entwickelnde Konfigurationssystem genutzt werden.

Im Kapitel 18 werden Beschreibungslogiken (engl. Description Logics, DL) anhand der grundlegenden Sprache $\mathcal{ALC}$ und die dieser Arbeit zugrunde liegenden Sprache des DL-Systems *NEOCLASSIC* vorgestellt. In den folgenden Abschnitten wird die Formalisierung der Komponenten auf Basis der Beschreibungssprache $\mathcal{ALC}$ formalisiert.

9.2.3 Spezielle Anforderungen an die Softwarekonfiguration

In diesem Abschnitt sollen nun die Anforderungen für das Konfigurationsmanagement von Softwarekomponenten zusammengefasst werden. Softwarekomponenten werden in dieser Arbeit mittels der Sprache SCDL beschrieben. Um Softwarekomponenten von den Komponenten der Konfiguration unterscheiden zu können, wird im Folgenden, wenn der Zusammenhang nicht deutlich ist, von *S*- (Softwarekomponenten) bzw. von *K-Komponenten* (Komponenten im Sinne der Konfiguration) gesprochen.

Aus Sicht der Konfiguration müssen folgende Besonderheiten von S-Komponenten berücksichtigt werden:

1. Wie in Abschnitt 1.3 und Kapitel 7 dargestellt wurde, werden S-Komponenten in atomare und komplexe Komponenten unterteilt. Komplexe S-Komponenten werden gebildet, indem Komponenten-Frameworks mit anderen (atomaren oder komplexen) Komponenten

zusammengesetzt werden. Pro Slot eines Komponenten-Frameworks kann es eine Menge alternativer Komponenten geben, die den Kontrakt eines Slots erfüllen. Die alternativen Komponenten eines Slots können Abhängigkeiten zu den alternativen Slots anderer Komponenten besitzen. Sowohl die S-Komponenten als auch deren Abhängigkeiten müssen in einem Systemmodell dargestellt werden.

2. Eine S-Komponente wird in Bezug auf ihren Anwendungsbereich durch Annotationen näher beschrieben (interpretiert, siehe Abschnitt 8.2.1). Annotationen werden einer Komponente durch die Abbildungen aus $\mathcal{K}$, $\mathcal{A}$ und $\mathcal{Z}$ (siehe Abschnitt 8.3) zugeordnet. Da eine Komponenten innerhalb des Systemmodells unterschiedlich interpretiert werden kann, müssen diesen Komponenten auch jeweils unterschiedliche Annotationen zugeordnet werden können.

3. Jede Komponente besitzt eine Reihe von Prozeduren, die von außen, also von einem Client, der außerhalb des konfigurierten Systems steht, aufgerufen werden dürfen. Der Umfang dieser Prozeduren kann von Konfiguration zu Konfiguration differieren. Die aufrufbaren Prozeduren einer Komponente müssen deshalb konfigurationsspezifisch (siehe Abschnitt 10.2.1) angegeben werden können.

Da sowohl S-Komponenten als auch die zugeordneten Abbildungen einer Terminologie und die aufrufbaren Prozeduren einer Komponente Bestandteile einer konkreten, im Einsatz befindlichen Komponenteninstanz sind, erscheint es sinnvoll, sie auch jeweils als Komponenten im Sinne einer Konfiguration (K-Komponenten) zu betrachten.

Definition:

Eine *K-Komponente* ist

- eine S-Komponente,
- eine Abbildung aus $\mathcal{K}$, $\mathcal{A}$ oder $\mathcal{Z}$, oder
- eine Menge von Prozeduren P.

Eine *komplexe K-Komponente* ist eine geordnete Liste von Subkomponenten (S-Komponenten). Alle diese Bestandteile, also S-Komponenten, Abbildungen und Prozedurmengen, können auch wieder abstrakt, also im Sinne eines UND/ODER-Graphen ein ODER-Knoten sein.

Beispiel:

In Abbildung 9.1 ist ein Anwendungssoftwareprodukt `ASP` dargestellt, das aus zwei S-Komponenten besteht, die über die Verweise `k` und `f` referenziert werden. Die Komponente `k` ist entweder eine atomare Komponente `A` oder `B`. Für die von `f` referenzierte Komponente steht nur eine Komponente zur Wahl. Jede Komponente – für die Komponente `B` wurde auf

die Darstellung verzichtet – hat drei Attribute: So sind die Attribute der Komponente A mit Abbildungen K_A, A_A und Z_A sowie einer Menge P_A und die Attribute der Komponente X mit den Werten von O2 (K_{X1} oder K_{X2}), A_X und Z_X sowie der Menge P_X belegt. Das Softwareprodukt ASP kann nun aus den Komponenten X und A (oder B) zusammengesetzt werden. Die Terminologie ist für A fest vorgegeben. Für die Komponente X kann der Konfigurierer zwischen zwei Abbildungen für die Annotationen von Signaturen wählen.

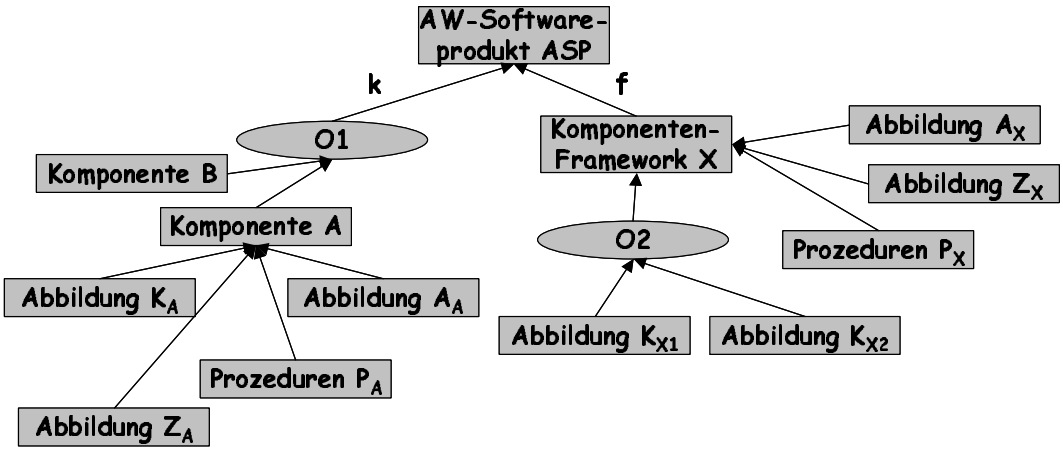


Abbildung 9.1: Beispiel-Systemmodell

9.2.4 Anforderungen durch den Konfigurationsprozess

Neben den Anforderungen, die sich direkt aus des im Rahmen der Konfiguration darzustellenden Gegenstandsbereichs (hier der Softwarekomponenten) ergeben, spielt der Konfigurationsprozess ein wesentliches Kriterium bei der Auswahl eines geeigneten Repräsentationsformalismus. Es stellt sich also die Frage, wie der Konfigurationsprozess überhaupt definiert soll und wie er möglichst gut unterstützt werden kann.

Wie in Abschnitt 1.5 erklärt wurde, soll die Konfiguration anhand von Beschreibungen von Geschäftsvorfällen (Geschäftsprozessbeschreibungen, -modellen) vollzogen werden können. Dazu werden nach und nach typische Interaktionen der Komponenten zu typischen Geschäftsprozessbeschreibungen zusammengesetzt. Ein Problem ist nun, dass das Systemmodell keine Informationen über das Verhalten einer S-Komponente hat, da die internen Informationen der Komponente (Prozeduren und Verhalten) nicht in das Systemmodell mit einfließen sollen. Aus diesem Grund wird eine Entscheidungskomponente benötigt, die alle aufgrund der augenblicklichen Konfiguration möglichen Interaktionen berechnet, darauf basierend den Benutzer (Konfigurierer) eine Interaktion auswählen lässt und dann die über die Interaktion ausgewählte Komponente an das Konfigurationssystem zurück liefert. Ferner muss er die Möglichkeit haben, konstruierte Geschäftsprozesse – und damit die gefällten Konfigurationsentscheidungen – rückgängig zu machen. Auf dieses Zusammenspiel wird in Kapitel 10 genauer eingegangen.

Um einen Konfigurationsprozess überhaupt durchführen zu können, werden geeignete Operatoren benötigt. Mittels dieser Operatoren ist es dann möglich, neue Konfigurationen zu erzeugen, Komponenten zu einer Konfiguration hinzuzufügen und die Angaben zu einer Konfiguration zu konkretisieren, indem der Typ einer Komponente genauer spezifiziert wird.

Da ein Konfigurierer (siehe Abschnitt 1.2) sich bei bestimmten Konfigurationsentscheidungen auch einmal irren kann oder potentielle Konfigurationsentscheidungen ausprobieren möchte, muss die Möglichkeit bestehen, Entscheidungen, die einmal gefällt wurden, wieder rückgängig zu machen bzw. zu korrigieren (Belief Revision).

Insgesamt werden dann mindestens folgende Operatoren benötigt:

- `newConfiguration` zum Erzeugen einer neuen Konfiguration,
- `addComponent` zum Erzeugen einer neuen Komponente innerhalb einer Konfiguration,
- `delComponent` zum Löschen einer Komponente innerhalb einer Konfiguration,
- `addReference` zum Einfügen einer konstituierenden Komponente zu einer komplexen Komponente,
- `delReference` zum Löschen einer Referenz zwischen einer konstituierenden und einer komplexen Komponente,
- `specializeComponent` zum Spezialisieren einer Komponente,
- `generalizeComponent` zur Rücknahme von Spezialisierungsinformationen zu einer Komponente und

`isBound` zum Testen, ob es sich bei einer Konfiguration um eine gebundene, d.h. tatsächlich nutzbare Konfiguration handelt oder nicht.

9.3 Die Konfigurationsbeschreibungssprache CCL

In diesem Abschnitt werden nun Syntax (Unterabschnitt 9.3.4) und Semantik (Unterabschnitt 9.3.5) der Konfigurationssprache CCL (Component Configuration Language) definiert. Zuvor allerdings wird ein einführendes Beispiel präsentiert, das in der allgemeinem Beschreibungslogik $\mathcal{ALC}$ (Unterabschnitt 9.3.1) und danach (Unterabschnitt 9.3.3) in der speziellen CCL-Syntax formuliert wird, um ein intuitives Verständnis für diese Sprache und ihre Semantik zu schaffen.

9.3.1 Einführendes Beispiel

Das Beispiel aus Abschnitt 7.1 wird erweitert: Die Komponenten `client` und `server` werden dazu in `client1` und `server1` umbenannt. Zusätzlich werden zwei weitere

Komponenten `client2` und `server2` definiert, die ebenfalls die Kontrakte `clientC` und `serverC` einhalten.

Nun wird das Anwendungssoftwareprodukt ASP – eine komplexe Komponente – betrachtet, das aus mehreren Komponenten zusammengesetzt werden muss. Dazu werden ein `csFramework` und zwei Komponenten benötigt, die aus Kompatibilitätsgründen die Kontrakte `clientC` und `serverC` einhalten müssen.

Zuerst werden dazu die Kompatibilitätsbeziehungen in Form eines Konzeptgraphen (siehe Abbildung 9.2) dargestellt. Zur Unterscheidung der wesentlichen Elemente der komponentenbasierten Softwareentwicklung werden die drei Konzepte `CONTRACT`, `COMPONENT` und `COMPLEX` eingeführt. Demnach sind `clientC` und `serverC` Kontrakte und alle anderen Elemente Komponenten, von denen einige diese Kontrakte einhalten. Die Komponente ASP ist komplex, d.h. sie wird aus anderen Komponenten zusammengesetzt. In einer Beschreibungslogik lassen sich diese Sachverhalte durch Konzeptdefinitionen (siehe Abschnitt 18.2) wie folgt darstellen.

```
CONTRACT  ⊑ TOP
COMPONENT ⊑ TOP
COMPLEX ⊑ COMPONENT
serverC  ⊑ CONTRACT
clientC  ⊑ CONTRACT
server1  ⊑ (COMPONENT ⊓ serverC)
server2  ⊑ (COMPONENT ⊓ serverC)
client1  ⊑ (COMPONENT ⊓ clientC)
client2  ⊑ (COMPONENT ⊓ clientC)
ASP ⊑ COMPLEX
csFramework ⊑ COMPONENT
```

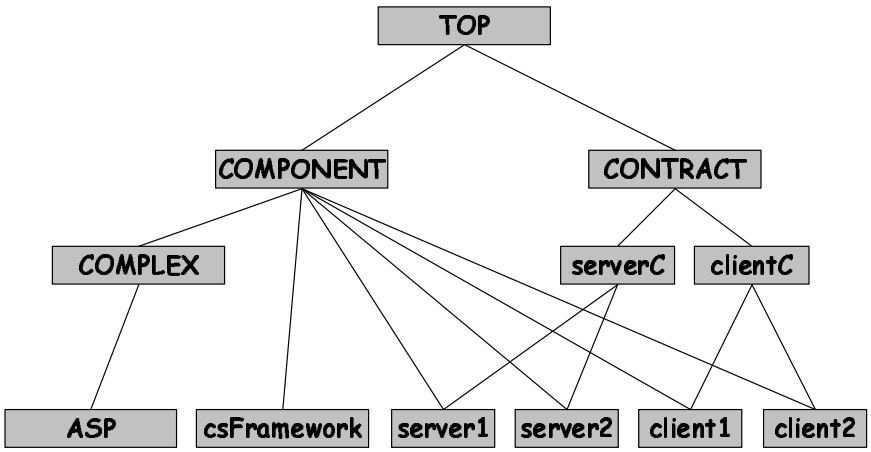


Abbildung 9.2: Subordinationsbeziehungen

Komponenten-Frameworks werden durch andere Komponenten parametriert. Dazu weisen sie Slots auf, in die die anderen Komponenten eingesetzt werden. Die Komponenten müssen vorgegebenen Kontrakten entsprechen, damit sie reibungslos interagieren können. Auch die Informationen, welche Slots (mit welchen Kontrakten) eine Komponente besitzt, müssen in einem Konfigurationssystem beschrieben werden. Diese Parameterbeziehungen können als Rollen zu den entsprechenden Konzepten (Kontrakten) definiert werden. Graphisch lässt sich das für den einzigen Komponenten-Framework des Beispiels durch einen Graphen (siehe Abbildung 9.3) veranschaulichen. Der Framework `csFramework` hat zwei Slots, die über die Bezeichner `s` und `c` referenziert werden und die durch die Kontrakte `serverC` bzw. `clientC` typisiert sind. In einer Beschreibungslogik lässt sich dieser Sachverhalt durch Attribute bzw. Rollen formulieren. `s` (bzw. `c`) ist eine Rolle, die als binärer Beziehungstyp zwischen den Konzepten `csFramework` und `serverC` (bzw. `clientC`) definiert ist[1].

$$s \sqsubseteq (csFramework \times serverC)$$
$$c \sqsubseteq (csFramework \times clientC)$$

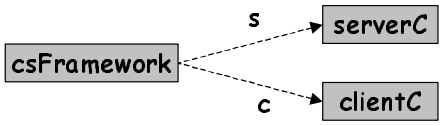


Abbildung 9.3: Parametrierungsbeziehungen

Die für einen Anwendungsbereich (z.B. Branche) als sinnvoll erachteten Konfigurationsmöglichkeiten einer Menge von Komponenten müssen nun im Sinne eines zusammenhängenden Anwendungssoftwareprodukts definiert werden. Dazu können die möglichen Komposita, wie sie schon in Abschnitt 3.4 exemplarisch dargelegt wurden, graphisch als *UND-Knoten* repräsentiert werden. In Abbildung 9.4 wird dazu das Anwendungssoftwareprodukt `ASP` konstruiert, das aus drei Komponenten zusammengesetzt wird. Im Beispiel ist eine Komponente konkret vorgegeben (`csFramework`). Die beiden anderen Komponenten können erst einmal beliebig sein. Jedoch wird durch zusätzliche Bedingungen (Constraints) gefordert, dass die Komponenten auf die durch die Rollen `c1` und `c2` verwiesen wird, identisch mit den Komponenten `s` bzw. `c` sein müssen, die in die Slots `s` bzw. `f` der Komponente `csFramework` gesetzt werden.

In einer Beschreibungslogik (zur Schreibweise siehe in Teil V Kapitel 18) lässt sich dieser Sachverhalt folgendermaßen formulieren:

---

[1] In den meisten abstrakten Syntaxdefinition, so auch in *ALC*, werden Rollen nicht explizit definiert, sondern innerhalb einer Konzeptbeschreibung eingeführt. Hier wird jedoch zur Darstellung das Teilmengensymbol über dem kartesischen Produkt gewählt.

```
f  ⊑ (ASP × csFramework)
c1 ⊑ (ASP × COMPONENT)
c2 ⊑ (ASP × COMPONENT)
ASP = ASP' ⊓ (s∘f ↓ c1) ⊓ (c∘f ↓ c2)
```

Es werden drei Rollen `f`, `c1` und `c2` definiert, die auf die Subkomponenten des Anwendungssoftwareprodukts verweisen. Die Menge der gültigen Konfigurationen ist nun zum einen durch die Struktur dieser drei Komponenten vorgegeben, und durch die Constraints wird zusätzlich gefordert, dass die Komponenten `s∘f` und `c1` bzw. `c∘f` und `c2` identisch sind und somit den entsprechenden Kontrakten entsprechen müssen. Damit sind nun die Konfigurationsalternativen für dieses kleine Beispiel vollständig beschrieben.

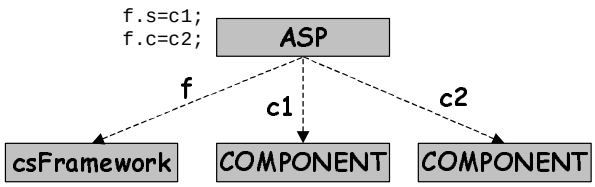


Abbildung 9.4: Kompositionsbeziehungen

Um eine Konfiguration des Anwendungssoftwareprodukts ASP zu erstellen, müssen konkrete Extensionen – im Bereich der Beschreibungslogiken werden diese als *Individuen* bezeichnet – erzeugt werden. Ein Individuum repräsentiert eine K-Komponente, die innerhalb des Anwendungssoftwareprodukts eingesetzt werden soll. Welche Komponente mit welchen anderen Komponenten parametriert und dann zum Aufbau komplexer Komponenten herangezogen werden soll, wird durch Attribut- bzw. Rollenbeziehungen zwischen diesen Individuen angegeben. Ein System für Beschreibungslogiken (DL-System) überwacht nun die Gültigkeit einer Konfiguration anhand der zuvor angegeben intensionalen Konfigurationsbeschreibung. Diese verbietet z.B., dass in den Framework `csFramework` Komponenten eingesetzt werden, die nicht die Kontrakte `clientC` bzw. `serverC` einhalten.

9.3.2 Ergebnis der Konfiguration

Wie oben dargestellt wurde, werden für den Konfigurationsprozess verschiedene Typen von Beziehungen (Subordination, Komposition, Parametrierung) zwischen Komponenten (und Kontrakten) definiert. Zwei Fragen blieben bislang noch offen: Wie wird das Ergebnis eines Konfigurationsprozesses überhaupt repräsentiert, und durch welche Operationen kann aus dem Konfigurationsraum eine konkrete Konfiguration abgeleitet werden?

Beschränkt man sich auf das angestrebte Ergebnis einer Konfiguration, dann interessiert den Konfigurierer letztendlich nur, welche Komponenten mit welchen anderen Komponenten

zusammengesetzt werden müssen, also die im Rahmen der Konfiguration tatsächlich ausgewählten Kompositionsbeziehungen. Da die Kompositionsbeziehungen immer hierarchisch und azyklisch sind, lässt sich das Ergebnis einer Konfiguration immer als UND-Baum (UND/ODER-Baum ohne ODER-Knoten) darstellen. In [Bach 1994] wird dazu eine Mengenschreibweise vorgeschlagen, in der von der Wurzel beginnend alle Pfade über die Attribute der komplexen Komponenten angegeben werden und zusätzlich die Komponenten entsprechend ihres Typs durch eine Abbildung γ bezeichnet werden. Die oberste Komponente ist nicht Attribut einer anderen Komponente und wird deswegen mit „ε“ referenziert. So lässt sich beispielsweise die (noch abstrakte) Konfiguration aus Abbildung 9.4 durch die Menge M und die Abbildung γ (hier als Menge von Tupeln) folgendermaßen beschreiben:

```
M    = {ε, ε.f, ε.c1, ε.c2}

γ    = {  (ε, ASP),
          (ε.f, csFramework),
          (ε.c1, COMPONENT),
          (ε.c2, COMPONENT)}
```

In dieser Arbeit werden diese beiden Aspekte in einer Darstellung vereinigt, indem jedem Pfad der entsprechende Typ angehängt wird.

Definition Konfiguration

Eine Konfiguration K ist eine Teilmenge aus dem kartesischen Produkt der Pfade $\mathcal{L}^*$ über die Attributbezeichner $\mathcal{L}$ komplexer Komponenten und der Menge der Bezeichner der Komponenten L(<PATH>).

$K \subset (\mathcal{L}^* \times L(<PATH>))$.

Als Schreibweise kann die Mengenschreibweise beibehalten werden, wobei die Komponente durch einen Doppelpunkt getrennt angehängt wird. Das Beispiel würde dann folgendermaßen definiert werden:

```
M'   = {  ε:ASP, ε.f:csFramework, ε.c1:COMPONENT, ε.c2:COMPONENT }
```

9.3.3 Definition von CCL

In diesem Abschnitt wird nun die Sprache CCL (Component Configuration Language) zur Beschreibung von Konfigurationen von Softwarekomponenten eingeführt. Dabei werden im Wesentlichen die zuvor dargestellten drei Aspekte – Subordination (vgl. Begriff Substituierbarkeit in Abschnitt 4.1.2), Parametrierungsbeziehungen und Komposition – berücksichtigt. Die Substituierbarkeitsbeziehungen zwischen Komponenten sind logisch nichts anderes als Parametrierungsbeziehungen, denn beide können durch Rollenbeziehungen

in der zugrunde liegenden Beschreibungslogik repräsentiert werden. Im weiteren Verlauf wird auf die Konfiguration der Terminologie und die Mengen aufrufbarer Prozeduren nicht eingegangen, da diese Aspekte genauso wie die Konfiguration der Softwarekomponenten behandelt werden können.

In CCL gibt es ein zentrales Sprachkonstrukt (`concept`), das zur Beschreibung von Softwarekomponenten und der Beziehungen (Kompositions- und Parametrierungsbeziehungen) zwischen diesen Komponenten dient. Als Konzepte werden Elemente der komponentenbasierten Softwareentwicklung, nämlich S-Komponenten (Komponenten oder Komponenten-Frameworks), Kontrakte oder komplexe S-Komponenten, die aus anderen S-Komponenten zusammengesetzt werden, abgebildet. Unter Hinzunahme der Substituierbarkeitsbeziehungen, die hier mittels der Subsumtion ausgedrückt wird, ergibt sich so die Menge aller Konfigurationen (*Konfigurationsraum*) zu einem Anwendungssoftwareprodukt. Konfigurationsalternativen kommen dadurch zustande, dass verschiedene Komponenten von demselben Kontrakt subsumiert werden. Um die verschiedenen Typen von Softwareelementen auseinanderhalten zu können, müssen sie durch Angabe spezieller Schlüsselwörter (`CONTRACT`, `COMPONENT` und `COMPLEX`) von den entsprechend vorgegebenen Konzepten abgeleitet werden. Komponenten-Frameworks unterscheiden sich von einfachen S-Komponenten dadurch, dass zusätzlich Slots angegeben werden, in die Parameterkomponenten eingesetzt werden. Komplexe Komponenten besitzen ebenfalls wie die Komponenten-Frameworks Attribute. Jedoch handelt es sich hier nicht um Parameter, sondern um die konstituierenden S-Komponenten der komplexen Komponente. Sie haben also eine andere Bedeutung, nämlich die der Komposition. Zusätzlich können zu einer komplexen Komponente Konsistenzbedingungen (`Constraints`) angegeben werden, die z.B. die Auswahlmöglichkeiten der konstituierenden Komponenten einschränken. Genau ein Konzept muss von dem Konzept `MAIN` subsumiert werden. Dieses Konzept kennzeichnet das Hauptelement des Konfigurationsraumes. Es darf also auch nur eine Instanz (Individuum) zu diesem Konzept geben. Ausgehend von diesem Individuum wird eine Konfiguration verwaltet.

Analog zum Beispiel werden im Folgenden die entsprechenden Konfigurationselemente in CCL definiert:

```
concept serverC derived CONTRACT {}
concept clientC derived CONTRACT {}
concept server1 derived serverC, COMPONENT {}
concept server2 derived serverC, COMPONENT {}
concept client1 derived clientC, COMPONENT {}
concept client2 derived clientC, COMPONENT {}
concept csFramework derived COMPONENT {
     s: serverC; /* parameter */
     c: clientC; /* parameter */
}
concept ASP derived COMPLEX, MAIN {
     f : csFramework;   /* constituent */
```

```
        c1: COMPONENT;     /* constituent */
        c2: COMPONENT;     /* constituent */
        constraint x1 (f.s = c1);
        constraint x2 (f.c = c2);
    }
```

9.3.4 Syntax von CCL

Die Syntax von CCL wird in Form einer EBNF-Grammatik (siehe Abschnitt 16.2) angegeben. Das Startsymbol der Grammatik ist <SPACE>. Die Menge aller gültigen Konfigurationen zu einer Menge vorgegebener Komponenten wird als Liste von Konzeptdefinitionen angegeben.

```
<SPACE>   ::=  { <CONCEPT> }+.
```

Ein Konzept hat einen Namen (<PATH>), der dem Komponentennamen aus der SCDL-Beschreibung der entsprechenden S-Komponente entspricht. Die Komponente stützt sich auf keinem, einem oder mehreren Kontrakten ab, die ebenfalls durch ihren SCDL-Pfad referenziert werden. Zusätzlich muss ein Basiskonzept angegeben werden. Der Körper der Konzeptdefinition enthält Informationen über die Parametrierungs- oder Kompositionsbeziehungen einer Komponente.

```
<CONCEPT> ::=  concept <PATH> derived {<PATH>,} <BASECONCEPT> <CONCEPTBODY>.
```

Alle Konzepte sind direkt einem Basiskonzept untergeordnet: entweder gehören sie zu den Kontrakten, zu den atomaren Komponenten oder zu den komplexen Komponenten. Genau eine Komponente gehört zusätzlich dem Konzept MAIN an. Sie kennzeichnet später den Wurzelknoten einer Konfiguration.

```
<BASECONCEPT> ::=  ( CONTRACT | COMPONENT | COMPLEX ) [, MAIN ].
```

Der Körper der Konzeptdefinition enthält Informationen über die Zusammensetzung einer Komponente. Handelt es sich um eine komplexe Komponente, so werden Attribute für die Konstituenten angegeben, handelt es sich um einen Komponenten-Framework, so werden die Slots als Attribute angelegt. Zusätzlich können Konsistenzbedingungen formuliert werden, die zwischen den Elementen, die von den Attributen referenziert werden, eingehalten werden müssen.

```
<CONCEPTBODY> ::=  "{" { <ATTRIBUTE> } { <CONSTRAINT> } "}".
```

Attribute werden durch einen Bezeichner und einen Pfadnamen eines anderen Konzeptes definiert.

```
<ATTRIBUTE>   ::=  <ID> : <PATH>.
```

Konsistenzbedingungen werden mit einem für das Konzept eindeutigen Namen bezeichnet und durch einen booleschen Ausdruck angegeben.

```
<CONSTRAINT>        ::= constraint <ID> <BOOLEXPR>.
```

Über boolesche Ausdrücke sind u.a. Co-Referenz-Bedingungen formulierbar. Prinzipiell lassen sich alle berechenbaren booleschen Ausdrücke formulieren. In NEOCLASSIC sind z.B. beliebige C++-Methoden einbindbar.

```
<BOOLEXPR>          ::= <COREFERENCE> |    /* Anderer Boolescher
                                              Ausdruck */.
```

Mittels Co-Referenz-Bedingungen wird formuliert, dass zwei Attribute das gleiche Individuum referenzieren.

```
<COREFERENCE>       ::= <ID>{"."<ID>} = <ID>{"."<ID>}.
```

Gültige Identifizierer sind alphanumerische Zeichenketten ohne Sonderzeichen.

```
<ID>                ::=       /* Gültiger Identifizierer */
```

Pfadausdrücke sind Folgen von Identifizierern, die durch Punkte („.“) voneinander getrennt sind.

```
<PATH>              ::= /* Bezeichner für eine Komponente/ein Kontrakt,
                           siehe Abschnitt 7.2 */
```

9.3.5 Semantik von CCL

Die Semantik von CCL wird durch eine Übersetzung in die $\mathcal{ALC}$-Beschreibungslogik (siehe Abschnitt 18.2) definiert. Das grobe Übersetzungsprinzip wurde schon in Abschnitt 9.3.1 angedeutet. Es soll hier nun formalisiert werden. Es werden alle Konzepte aus einer CCL-Beschreibung nacheinander übersetzt. Diese übersetzten Konzepte bilden dann die TBox (siehe 18.2) des beschriebenen Systems.

Für den Konfigurationsprozess ist es sinnvoll, eine möglichst maximale, abstrakte Ausgangskonfiguration zu berechnen, die für alle gebundenen Konfigurationen eines Anwendungssoftwareproduktes identisch ist. Diese Ausgangskonfiguration bildet dann die initiale ABox des Konfigurationssystems (siehe Abschnitt 9.3.6).

9.3.5.1 TBox

Die Übersetzung der intensionalen Beschreibung des Konfigurationssystems (Anwendungssoftwareprodukt) wird durch die Abbildung `CCL2DL` („CCL nach Description Logic“) angegeben. Wie in Kapitel 7 werden auch hier meist die Übersetzungsregeln für die einfache Anwendung – also z.B. für ein Konzept und nicht für eine Liste von Konzepten – angegeben. Um einen vollständigen Übersetzungsalgorithmus zu definieren, müssen die CCL-Beschreibungen einfach nur sequentiell nacheinander durch die hier angegebenen Regeln übersetzt werden.

Als erstes müssen die grundlegenden Konzepte von CCL (`COMPONENT`, `CONTRACT`, `COMPLEX` und `MAIN`) als Konzepte in der Beschreibungslogik definiert werden. Dieses „Wissen“ wird auch als *Komponentenwissen* bezeichnet.

```
CONTRACT ⊑ TOP
COMPONENT ⊑ TOP
COMPLEX ⊑ COMPONENT
MAIN ⊑ TOP
```

Ein CCL-Konzept wird als primitives Konzept der Beschreibungslogik definiert. Die dazugehörige Konzeptbeschreibung besteht aus der Konjunktion von Konzepten – einem vordefinierten Konzept und/oder einem oder mehreren in CCL definierten Konzepten –, die alle „1:1“ in der Beschreibungslogik abgebildet werden können. Ferner enthält die Konzeptbeschreibung die Attribut- und Constraint-Definitionen für komplexe Komponenten.

Die Regel für die Übersetzung einer Konzeptdefinition ist dann folgendermaßen definiert:

```
CCL2DL( concept <PATH> derived <PATH_1> ... <PATH_n>
          <ATTRIBUTES> <CONSTRAINTS> ) =
     <PATH> ⊑ (<PATH_1>  ⊓ ... ⊓ <PATH_n>
                 ⊓ transAttrs(<ATTRIBUTES>)
                 ⊓ transConstraints(<CONSTRAINTS>)
              )
```

Gemäß dieser Abbildung werden CCL-Attribute als Rollen ($R_1, \ldots, R_n$) abgebildet, die jedoch immer nur genau eine Extension beinhalten dürfen. Sie werden also als einelementige Mengen (für die Kardinalität c der Menge gilt: $1 \geq c \geq 1$) abgebildet.[1] Die Attributdefinitionen werden durch die Übersetzungsfunktion `transAttrs` erzeugt:

```
transAttrs( R_1:C_1; ...R_n:C_n; ) =
          ∀ R_1:C_1 ⊓ ≥ 1 R_1 ⊓ ≤ 1 R_1
          ⊓ ... ⊓
          ∀ R_n:C_n ⊓ ≥ 1 R_n ⊓ ≤ 1 R_n
```

Die Constraints werden als boolesche Ausdrücke verstanden, die ein Konzept genauer definieren bzw. aus extensionaler Sicht die Menge der Extensionen eines Konzeptes weiter einschränken. Die Beschreibungen der Constraints werden durch die Abbildung `transConstraints` erzeugt:

---

[1] In Kapitel 18 werden auch Attribute für Beschreibungslogiken eingeführt. Da in NEOCLASSIC keine Attribute zur Verfügung stehen, müssen Rollen dafür genutzt werden, indem man die Kardinalität der Rollen auf eins beschränkt.

transConstraints(constraint <ID$_1$> <BT$_1$>; ... constraint <ID$_n$> <BT$_n$>;)=
transBoolExpr(<BT$_1$>) ⊓ ... ⊓ transBoolExpr(<BT$_n$>)

Die booleschen Ausdrücke werden durch eine Konjunktion miteinander verknüpft. Wie ein einzelner Term in die Beschreibungslogik übersetzt wird, wird hier nicht genauer definiert. Die Ausdrucksmächtigkeit von CCL und damit die Übersetzung sind von der konkret gewählten Beschreibungslogik abhängig. Im DL-System *NEOCLASSIC*, das der Implementierung dieser Arbeit (siehe dazu auch Kapitel 18) zugrunde liegt, können Bedingungen ausschließlich als C++-Methoden definiert werden.

Eine einfache, aber immer wieder benötigte Konsistenzbedingung in Konfigurationssystemen ist die Gleichheit von Konfigurationselementen (Co-Referenz, siehe [Bach 1994] und Abschnitt 18.2 im Anhang). Sie wird in konkreten Konfigurationssystemen als Gleichheit von Individuen (Komponenten) realisiert, die über Pfadausdrücke von Attributbezeichnern identifiziert werden. Sie wird in CCL folgendermaßen übersetzt:

transBoolExpr(<ID$_{A1}$>.<ID$_{A2}$> ... .<ID$_{An}$> = <ID$_{B1}$>.<ID$_{B2}$> ... .<ID$_{Bm}$>) =
<ID$_{An}$> ∘...∘ <ID$_{A1}$> ↓ <ID$_{Bm}$> ∘...∘ <ID$_{B1}$>

<ID$_{A1}$>.<ID$_{A2}$> ... .<ID$_{An}$> bzw. <ID$_{B1}$>.<ID$_{B2}$> ... .<ID$_{Bm}$> sind Pfade von Rollenbezeichnern, über die Konfigurationselemente wie z.B. Subkomponenten oder Parameterkomponenten von anderen Komponenten referenziert und zueinander in Beziehung – in diesem Fall die Identitätsbeziehung – gesetzt werden können.

9.3.6 Konfigurationsoperatoren

Während CCL die Komponenten einer Konfiguration nur auf intensionaler Ebene beschreibt, werden nun zusätzliche Operatoren benötigt, die das Bearbeiten und Manipulieren einer konkreten Konfiguration ermöglichen. Dazu werden die in Abschnitt 9.2.4 angesprochenen Operatoren benötigt. Sie sollen im Folgenden genauer beschrieben und anhand eines Beispiels verdeutlicht werden.

Die Operatoren arbeiten auf Individuen der durch eine CCL-Beschreibung definierten Komponenten. Im Sinne der Konfigurationsaufgabe würde man von Komponenteninstanzen[1] sprechen. Im Sinne einer anzustrebenden (gebundenen) Konfiguration sind diese Ausprägungen mit Kopien von Softwareprodukten (z.B. in Form von Softwarebibliotheken) oder ihren Beschreibungen (z.B. in Form SCDL-Beschreibungen) zu vergleichen. D.h. für jede gebundenen Konfiguration, die tatsächlich zum Einsatz kommen soll, werden Kopien der zugehörigen Bibliotheken bzw. Beschreibungen der Komponenten benötigt.

---

[1] Diese dürfen nicht mit Laufzeitinstanzen von Softwarekomponenten, wie sie in Abschnitt 3.2.9 beschrieben werden, verwechselt werden.

Zur Arbeit mit diesen Komponenteninstanzen – im Folgenden wird, um Verwechselungen vorzubeugen, nur von Individuen die Rede sein – steht in DL-Systemen die ABox (siehe Abschnitt 18.2) zur Verfügung. In ihr können Individuen und Rollenbeziehungen zwischen Individuen definiert werden.

In Abbildung 9.5 ist ein Graph abgebildet, der den Konfigurationsraum des Beispiels aus Abschnitt 9.3.1 darstellt. Die Abbildung soll zur Veranschaulichung der Konfigurationsoperatoren dienen. Das Anwendungssoftwareprodukt ASP setzt sich aus drei Komponenten zusammen: einem Komponenten-Framework und zwei anderen Komponenten, die in die Slots (`s` und `c`) des Frameworks passen müssen. Über die Subsumtionsbeziehung lässt sich ermitteln, dass `client1` und `client2` Komponenten sind und auch in den Slot `c` passen, da sie den notwendigen (aber hier nicht dargestellten) Kontrakt `clientC` einhalten. Genauso kann ermittelt werden, dass `server1` und `server2` Komponenten sind und auch in den Slot `c` passen, da sie den notwendigen (aber hier ebenfalls nicht dargestellten) Kontrakt `serverC` einhalten.

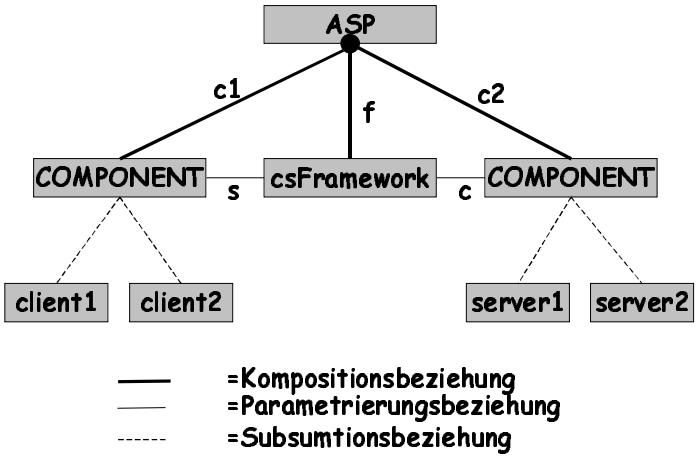


Abbildung 9.5: Graph des Konfigurationsraums (Beispiel)

9.3.6.1 Erzeugung neuer Konfigurationen: `newConfiguration`

DL-Systeme erzeugen nicht selbständig Individuen. Möchte man eine Konfiguration konstruieren, so müssen Individuen für die benötigten Komponenten erzeugt werden. Bei der Konfiguration eines Anwendungssoftwareprodukts erscheint es sinnvoll, mit der übergeordneten Komponente, die alle anderen Komponenten enthält, anzufangen. Diese Komponente muss in der CCL-Beschreibung durch das spezielle Konzept `MAIN` (siehe Abschnitte 9.3.4 und 9.3.5) gekennzeichnet werden. Der Operator `newConfiguration` erzeugt dann aus einer CCL-Beschreibung eine Menge von assertionalen Axiomen, die ein entsprechendes Individuum beschreiben.

Statt eine Konfiguration in der in Abschnitt 9.3.2 dargestellten Form anzugeben, wird bei der Semantikbeschreibung zunächst auf die Darstellung in der Beschreibungslogik $\mathcal{ALC}$ abgezielt, da in ihr – bzw. in der ihr ähnlichen Sprache NEOCLASSIC – später auch die Implementierung erfolgt. Demnach ist eine Konfiguration die Menge aller assertionalen Axiome (ABox), die durch die im Folgenden dargestellten Operatoren beeinflusst wird. Die in Abschnitt 9.3.2 präsentierte Darstellung ist nur als Ergebnis interessant, da sie für die Konfiguration eines Anwendungssystems als Eingabe benötigt wird.

Sei $\mathcal{L}$ die Menge der gültigen Individuenbezeichner, $\mathcal{C}$ die Menge aller $\mathcal{ALC}$-Konzeptbeschreibungen, die aus CCL-Konfigurationsbeschreibungen erzeugt werden können, und $\mathcal{X}$ die Menge der assertionalen Axiome der Beschreibungslogik $\mathcal{ALC}$ (siehe Abschnitt 18.2). Dann ordnet `newConfiguration` einer übersetzten CCL-Beschreibung eine (einelementige) Menge von assertionalen Axiomen (Teilmenge der Potenzmenge $\mathcal{P}(\mathcal{X})$ von $\mathcal{X}$) in $\mathcal{ALC}$ zu.

```
newConfiguration: 𝒞 → 𝒫(𝒳)
newConfiguration(C) = { (i:c) |    c ∈ C
                              ∧ MAIN subsumiert c
                              ∧ i ∈ ℒ ist innerhalb der Konfiguration eindeutig}
```

Analog zum Beispiel würde das Individuum `i-asp` $\in \mathcal{L}$ des Anwendungssoftwareprodukts `ASP` erzeugt werden. Dazu wird folgendes assertionales Axiom[1] durch den Operator `newConfiguration` berechnet:

```
i-asp : ASP       // (createIndividual i-asp ASP)
```

9.3.6.2 Erzeugen einer Komponente: `addComponent`

Der Operator `addComponent` erzeugt eine neue Komponente und fügt sie einer bestehenden Konfiguration – also einer Menge bestehender assertionaler Axiome – hinzu.

```
addComponent: 𝒫(𝒳) → 𝒫(𝒳)
addComponent(X)  = X ∪ { (i:COMPONENT) }
```

Im Beispiel werden durch den Konfigurierer drei Komponenten definiert:

```
i-c1 : COMPONENT        // (createIndividual i-c1 COMPONENT)
i-c2 : COMPONENT        // (createIndividual i-c2 COMPONENT)
```

---

[1] Im Folgenden werden neben der abstrakten Syntax hinter dem Kommentarzeichen „`//`“ zusätzlich die NEOCLASSIC-Befehle angegeben. Sie sind für die Implementierung des zu entwickelnden Konfigurationssystems relevant.

```
i-f  : COMPONENT        // (createIndividual i-f  COMPONENT)
```

9.3.6.3 Löschen einer Komponente: delComponent

Der Operator `delComponent` löscht eine Komponente aus einer bestehenden Konfiguration, indem alle assertionalen Axiome gelöscht werden, die sich auf dieses Individuum beziehen. (Der Unterstrich „_“ steht hier als „Wild Card“ für ein beliebiges Individuum oder Konzept).

$$\text{delComponent}: \mathcal{P}(\mathcal{X}) \times \mathcal{L} \rightarrow \mathcal{P}(\mathcal{X})$$
$$\text{delComponent}(X,i) = \{a \in X \mid a \neq (i:_) \wedge a \neq (_\ R\ i) \wedge a \neq (i\ R\ _)\}$$

9.3.6.4 Hinzufügen einer Komponente zu einer komplexen Komponente: addReference

Der Operator `addReference` fügt eine konstituierende Komponente zu ihrer komplexen Komponente hinzu. Als assertionales Axiom wird hierfür eine Rollenbeziehung zwischen komplexer und konstituierender Komponente benötigt.

$$\text{addReference}: \mathcal{P}(\mathcal{X}) \times \mathcal{L} \times \mathcal{L} \times \mathcal{R} \rightarrow \mathcal{P}(\mathcal{X})$$
$$\text{addReference}(X,i,j,r) = X \cup \{(i\ r\ j)\}$$

Für das Beispiel können die Rollenbeziehungen zwischen der komplexen (ASP) und ihren konstituierenden Komponenten gebildet werden. Die Rollenbeziehungen werden dann folgendermaßen angegeben:

```
i-asp c1  i-c1    // (addToldInformation i-asp (fills c1 i-c1))
i-asp c2  i-c2    // (addToldInformation i-asp (fills c2 i-c2))
i-asp f   i-f     // (addToldInformation i-asp (fills f  i-f))
```

9.3.6.5 Löschen einer Referenz zu einer komplexen Komponente: delReference

Der Operator `delReference` löscht eine Beziehung zwischen einer konstituierenden und einer komplexen Komponente. Aus der Menge der assertionalen Axiome wird die entsprechende Rollenbeziehung zwischen komplexer und konstituierender Komponente gelöscht.

$$\text{delReference}: \mathcal{P}(\mathcal{X}) \times \mathcal{L} \times \mathcal{L} \times \mathcal{R} \rightarrow \mathcal{P}(\mathcal{X})$$
$$\text{delReference}(X,i,j,r) = X \setminus \{(i\ r\ j)\}$$

9.3.6.6 Spezialisieren einer Komponente: specializeComponent

Wenn der Konfigurierer im Rahmen einer Konfigurationsentscheidung aus der Menge der möglichen Komponenten eine ausgewählt hat, so muss er das Individuum, das diese Komponente repräsentiert, mit dem Konzept spezialisieren.

```
spezializeComponent: 𝒫(𝒳) × ℒ × 𝒞 → 𝒫(𝒳)
spezializeComponent(X,i,c) = X ∪ {(i:c)}
```

Im Beispiel können die Individuen `i-c1, i-c2` und `i-f` konkretisiert werden. Z.B. könnte für `i-c1` die Komponente `client1`, für `i-c2` die Komponente `server2` und für `i-f` der Framework `csFramework` ausgewählt werden. Folgende assertionale Axiome drücken diese Wahl aus:

```
i-c1 : client1       // (addToldInformation i-asp (all c1 client1))
i-c2 : server2       // (addToldInformation i-asp (all c2 server2))
i-f  : csFramework   // (addToldInformation i-asp (all f csFramework))
```

9.3.6.7 Generalisieren einer Komponente: `generalizeComponent`

Wenn der Konfigurierer eine Konfigurationsentscheidung bzgl. der Spezialisierung einer Komponente rückgängig machen will, so müssen alle assertionalen Axiome, die die Komponente spezialisieren, aus der Konfiguration gelöscht werden.

```
generalizeComponent: 𝒫(𝒳) × ℒ → 𝒫(𝒳)
generalizeComponent(X,i) =  {a ∈ X | a ≠(i:_) ∧ a ≠(_ R i) and a ≠(i R _)}
                          ∪ {(i:COMPONENT)}
```

9.3.6.8 Testen, ob eine Konfiguration gebunden ist: `isBoundConfiguration`

Der Konfigurierer benötigt eine Testfunktion, um zu überprüfen, ob die von ihm bearbeitete Konfiguration noch abstrakt oder schon gebunden ist. Dazu muss überprüft werden, ob es Individuen gibt, die noch nicht ausreichend spezialisiert wurden, oder ob deren Konzepte Rollen aufweisen, die noch nicht besetzt sind.

```
isBoundConfiguration: 𝒫(𝒳) → {true, false}
isBoundConfiguration(X) = (∅ = { i | (i:c) ∈ X
                                ∧ (     i ist nicht spezieller als COMPONENT
                                    ∨   es existieren Attribute a von c
                                  für die gilt: (i a _) ∉ X)}
```

9.3.6.9 Konsistenzbedingungen

Die Co-Referenz-Constraints `x1` bzw. `x2`, die in Abschnitt 9.3.5 angegeben wurden und besagen, dass die Parameterkomponenten `s` bzw. `c` des Frameworks `f` gleich den

konstituierenden Komponenten `c1` bzw. `c2` von `ASP` sein sollen, können nun in der konkreten Konfiguration durch folgende Rollenbeziehungen ausgedrückt werden.

```
i-f c i-c1        // (addToldInformation i-f (fills c i-c1))
i-f s i-c2        // (addToldInformation i-f (fills s i-c2))
```

Das Inferenzsystem eines Beschreibungslogiksystems berechnet nach der Eingabe neuer Tatsachen (assertionalen Axiomen) automatisch die sich neu ergebenden Subsumtionsbeziehungen für die einzelnen Individuen. So kann ermittelt werden, welche Konzepte die Individuen `i-c1` und `i-c2` subsumieren, oder anders formuliert: welche S-Komponenten nun noch eingesetzt werden dürfen. In NEOCLASSIC steht dazu beispielsweise der Befehl `getParents` (siehe [Patel-Schneider et al. 1996]) zur Verfügung. Das Individuum `i-c1` wird von der Konjunktion der Konzepte `COMPONENT` und `clientC` und damit gleichzeitig von den Konzepten `client1` und `client2` subsumiert. D.h. für diese Komponente kommen sowohl `client1` als auch `client2` als mögliche Komponenten in Frage. Entsprechend gilt für das Individuum `i-c2`, dass es von der Konjunktion der Konzepte `serverC` und `COMPONENT` und damit gleichzeitig von den Komponenten `server1` und `server2` subsumiert wird.

Da nun alle Rollen besetzt sind – also keine Subkomponenten mehr fehlen – und alle Individuen nur noch von Komponenten subsumiert werden – also keine Komponente noch genauer definiert werden muss –, handelt es sich nun um eine gebundene Konfiguration.

Wie angedeutet, sind zusätzliche Abhängigkeiten zwischen den einzelnen Komponenten denkbar. So lässt sich der Fall konstruieren, dass nur zwei S-Komponenten für `i-c1` und `i-c2` eingesetzt werden dürfen, die die gleiche Nummer besitzen, also `client1` und `server1` oder `client2` und `server2`. Solche Constraints lassen sich in einfachen Beschreibungslogiken nicht formulieren. In NEOCLASSIC gibt es deswegen verschiedene Erweiterungen, die solche Formulierungen erlauben. Zum einen können mittels Regeln, die bei Subsumtion eines Individuums durch ein Konzept „feuern“, zusätzliche assertionale Axiome in die ABox eingefügt werden. Andererseits können Subsumtionen durch C++-Testfunktionen überprüft und abgelehnt werden (siehe dazu [Jaekel 1999] und [Resnick et al. 1996]). Dadurch können letztlich alle Constraints (Konsistenzbedingungen), die berechenbar sind, in NEOCLASSIC abgebildet werden.

10 Geschäftsprozessorientierte Konfiguration

Im Mittelpunkt dieses Kapitels steht die *geschäftsprozessorientierte Konfiguration* komponentenbasierter Anwendungssysteme. Im Rahmen einer solchen Konfiguration sollen die Anforderungen eines Anwenders auf Basis eines zuvor ausgewählten Anwendungssoftwareprodukts (siehe Kapitel 9) umgesetzt werden. Es ist zu beachten, dass es sich dabei ausschließlich um die Konfiguration auf Basis der *Komponentenbeschreibungen* (siehe Kapitel 7 und 8) handelt und dass somit das Ergebnis dieses Schritts nur eine Beschreibung (oder ein Entwurf) einer tatsächlichen Konfiguration darstellt, aber nicht die Konfiguration des Anwendungssystems selbst. Zudem werden ausschließlich funktionale, verhaltensbezogene und fachlich-terminologische Aspekte berücksichtigt, aber z.B. nicht nicht-funktionale Aspekte wie Performanz und Softwareergonomie.

Bevor man zu diesem Punkt der Anwendungssystemgestaltung kommt, müssen vom Anwender die Phasen der Anforderungsanalyse, der (systemunabhängigen) Soll-Konzeption und der Softwareauswahl abgeschlossen worden sein[1]. In der hier im Mittelpunkt stehenden Phase wird auf den Entwurf des zukünftigen Anwendungssystems abgezielt. Allerdings ist die hier vorgestellte Art der Konfiguration auch in früheren Phasen anwendbar, z.B. im Rahmen der Softwareauswahl, um zu prüfen, welches Produkt die Anforderungen des Anwenders am besten abdeckt.

Gemäß des hier erarbeiteten Vorschlags kann die Soll-Konzeptphase mit der Entwurfsphase enger verknüpft werden, indem die Soll-Geschäftsprozesse auf Basis der möglichen Geschäftsprozesse des Anwendungssoftwareprodukts, die formal durch die Traces ihrer Komponenten (Abschnitt 7.4) begründet sind, im Rahmen eines Konfigurationsprozesses parallel abgeleitet werden bzw. andersherum gesagt, die Konfigurationsentscheidungen während der Ableitung der Soll-Geschäftsprozesse getroffen werden.

Selbstverständlich kann nur ein Teil der Anforderungen und der daraus resultierenden Entwürfe auf Basis von Geschäftsprozessbeschreibungen formuliert werden. Andere Aspekte, die nicht in unmittelbaren Zusammenhang mit Geschäftsprozessen stehen, müssen durch andere Sprachen, z.B. ER-Diagramme oder natürliche Sprache, formuliert werden. Diese Aspekte werden nicht betrachtet.

---

[1] Für einen Vergleich einiger Vorgehensmodelle zur Einführung betrieblicher Anwendungssoftwareprodukte siehe z.B. [Kirchmer 1998].

10.1 Anforderungsvalidierung und -verifikation

Möchte man überprüfen, ob die mittels eines Anwendungssoftwareprodukts realisierbaren Geschäftsprozesse die geforderten *Soll-Geschäftsprozesse* erfüllen, so können verschiedene Wege eingeschlagen werden. In formalen Ansätzen werden die Anforderungen in Form der geforderten Geschäftsprozesse – dort dann als *Traces* bezeichnet – z.B. mittels einer Temporal- oder Trace-Logik (siehe z.B. [Olderog 1991]) und das konstruierte Anwendungssystem beispielsweise mit einem Prozesskalkül beschrieben. Darauf basierend kann unter bestimmten Voraussetzungen (z.B. durch Abstraktion auf einen endlichen Zustandsraum) maschinell überprüft werden, ob die Anforderungen durch das Anwendungssystem tatsächlich erfüllt werden. Dieser Weg der formalen Verifikation ist aufgrund der zu erwartenden Ungenauigkeit der formulierten Anforderungen eines Unternehmens und der Komplexität eines Anwendungssystems nicht praktikabel. Heute beherrschbare Techniken wie das Model Checking sind alleine aufgrund der Größe des Zustandsraums eines Anwendungssystems nicht oder nur auf einer sehr abstrakten Ebene anwendbar, wie u.a. in [Lechtenbörger 1997] gezeigt wird.

In der Praxis, z.B. im Rahmen einer R/3-Einführung, wird heute oft noch einzig und allein auf die Erfahrung der Anwendungsberater gesetzt, die „wissen“, welche Anforderungen sich durch ein Softwareprodukt abdecken lassen und welche nicht. Folglich wird bei der Umsetzung auf Basis dieses „Wissens“ versucht, die Geschäftsprozesse des Soll-Konzeptes direkt im Rahmen des Customizings umzusetzen. Dabei werden oft wechselseitige Beziehungen von Geschäftsprozessen unterschiedlicher Unternehmensbereiche ignoriert, was zu unnötigen Problemen und Verzögerungen bei der Konfiguration dieser Systeme führt. Ob das System den Anforderungen gerecht wird, kann dann erst im Nachhinein im produktiven Betrieb „getestet“ werden.

Status quo bei der Einführung betrieblicher Standardsoftwareprodukte ist der Entwurf auf Basis eines Softwarereferenzmodells (siehe Abschnitt 4.2.3), das u.a. eine Teilmenge der mit der Software realisierbaren Geschäftsprozesse beschreibt. Aus diesen Prozessen werden die für das Unternehmen gemäß des Soll-Konzeptes relevanten Prozesse ausgewählt, angepasst und danach im Rahmen des Customizings umgesetzt (siehe hierzu z.B. [Kirchmer 1998], [Keller und Teufel 1997]). Die Validierung der Anforderungen wird also im Vorfeld durchgeführt, indem Soll-Prozesse mit den angebotenen Geschäftsprozessen des Referenzmodells abgeglichen werden. Da diese Geschäftsprozesse aber nur exemplarischen Charakter haben, hat auch diese Art der Validierung nur einen begrenzten Nutzen. Z.B. kann nicht geprüft werden, ob die angepassten Geschäftsprozesse tatsächlich noch mittels der Software realisiert werden können.

Der Ansatz, der in diesem Kapitel vorgestellt wird, geht davon aus, dass Geschäftsprozesse aus Komponentenbeschreibungen, wie sie in Kapitel 7 erarbeitet wurden, abgeleitet werden.

Die möglichen Ableitungen werden durch einen in CCL vorgegeben Konfigurationsraum auf eine sinnvolle Teilmenge eingeschränkt. Kann der Konfigurierer oder Anwender einen Geschäftsprozess nicht ableiten, so wird er auch nicht von dem System unterstützt. Diese Idee wird in den weiteren Abschnitten vertieft.

Bei diesem Ansatz müssen jedoch gegenüber den rein formalen Ansätzen trotzdem Abstriche gemacht werden: Da es sich bei den SCDL-Beschreibungen um Unterspezifikationen (siehe Abschnitt 4.1.3) und damit Abstraktionen der konkreten Komponenten handeln kann, kann es sein, dass Prozesse, die sich ableiten lassen, tatsächlich mittels der gewählten Komponenten doch nicht umgesetzt werden können. **Die Ableitbarkeit ist also ein notwendiges, aber kein hinreichendes Kriterium.** Dieser Sachverhalt wird in Abbildung 10.1 veranschaulicht. Die Soll-Geschäftsprozesse eines Anwenders können durch ein Anwendungssystem nur teilweise abgedeckt werden. Allerdings bietet ein Anwendungssystem auch Geschäftsprozesse, die gar nicht benötigt werden. Die aus den Beschreibungen der Komponenten abgeleiteten Prozesse sind ausschließlich Soll-Geschäftsprozesse. Dafür hat der Konfigurierer zu sorgen, indem er seine abgeleiteten Prozesse mit den Soll-Geschäftsprozessen abgleicht. Allerdings lassen sich wegen der Unterspezifikation der Komponentenbeschreibungen gegenüber dem Anwendungssoftwareprodukt auch Geschäftsprozesse ableiten, die tatsächlich durch keine Konfiguration des Anwendungssoftwareprodukts unterstützt werden.

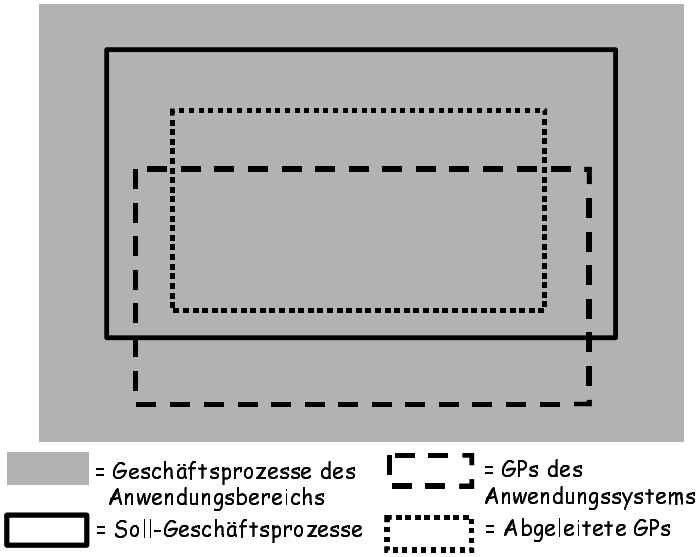


Abbildung 10.1: Soll-Geschäftsprozesse

Das Dilemma, indem man hier offensichtlich steckt, ist, dass ein Verzicht auf die Möglichkeit der Unterspezifikation zwar einen Gewinn bei der Genauigkeit der Beschreibungen bedeuten würde, jedoch gleichzeitig die Komplexität der Beschreibungen steigen, wodurch wiederum die Akzeptanz dieser Beschreibungen bei den Nutzern, also bei den Konfigurierern und Anwendern, sinken würde.

10.2 Zusammenhang von Konfiguration und Anforderungen

Im weiteren Verlauf dieses Kapitel soll nun beschrieben werden, wie die Konfiguration auf Basis von abgeleiteten Geschäftsprozessen vonstatten gehen soll. Grundidee ist, dass Geschäftsprozesse, die durch ein Anwendungssoftwareprodukt unterstützt werden, durch drei Aspekte bestimmt werden:

1. durch die Funktionalität, die durch die Komponenten definiert ist,
2. durch die möglichen Konfigurationen der Komponenten, die durch den Konfigurationsraum des Anwendungssoftwareprodukts vorgegeben sind, und
3. durch den Anwender, der eine gebundene Konfiguration aus dem Konfigurationsraum auswählt, die seinem Soll-Konzept und damit seinen Anforderungen möglichst weitgehend entspricht.

Die drei aufgezählten Aspekte – Funktionalität, Konfiguration und Anforderungen – und die daraus abzuleitenden Geschäftsprozesse stehen also in unmittelbarer Beziehung (siehe Abbildung 10.2) zueinander. Der Anwender hat Anforderungen an die Funktionalität eines Anwendungssoftwareprodukts bzw. der Komponenten. Andererseits schränkt die Funktionalität einer Standardsoftware die Realisierungsmöglichkeiten ein und nimmt damit Einfluss auf die tatsächlich umzusetzenden Anforderungen. Die Konfiguration hat Einfluss auf die Funktionalität und bedingt eine konkrete Funktionsweise der Komponenten. Die Geschäftsprozesse werden also durch die Funktionalität, die ausgewählten Parameter der Konfiguration und die Anforderungen des Anwenders bestimmt.

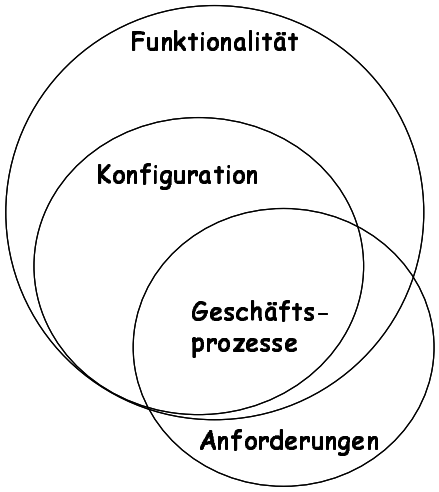


Abbildung 10.2: Wechselwirkungen mit Geschäftsprozessen

Aus extensionaler Sicht kann die Beschreibung von Geschäftsprozessen (oder ein Geschäftsprozessmodell) eines Unternehmens als eine Definition von Mengen von Traces verstanden werden, die aus den Beschreibungen der „Komponenten“ eines Unternehmens ableitbar sind (Ist-Zustand) bzw. sein sollen (Soll-Zustand). Zu diesen Komponenten eines Unternehmens zählen dann nicht nur die Softwarekomponenten, sondern auch die klassischen „Komponenten“ der Unternehmensorganisation, nämlich die organisatorisch festgelegten Abteilungen, Stellen usw. (Aufbauorganisation) inklusive ihrer Fähigkeiten, Aufgaben und Befugnisse (Ablauforganisation) sowie die zusätzlich benötigten Ressourcen wie Maschinen und Anlagen. Diese Dreiteilung in (Aufbau-)Organisation, Anwendungssysteme und Maschinen (inkl. Anlagen) entspricht der Dreiteilung der dritten Ebene der Unternehmensarchitektur nach Ferstl und Sinz (siehe Abschnitt 2.1.2).

In π-Kalkül-Schreibweise (siehe Kapitel 17) könnte man diesen Sachverhalt vereinfacht als paralleles Kompositum dreier Prozesse (oder Systeme) verstehen, die jeweils wieder als komplexe Prozessagenten definiert sind, und über entsprechende Schnittstellen miteinander kommunizieren und interagieren:

```
Unternehmen = (Aufbau-)Organisation | Anwendungssystem | Maschinen
```

Das Anwendungssystem setzt sich aus der Funktionalität des Anwendungssoftwareprodukts und der vorgenommenen Konfigurationseinstellungen zusammen:

```
Anwendungssystem = Anwendungssoftwareprodukt | Konfiguration
```

Das Geschäftsprozessmodell der zweiten Ebene der Unternehmensarchitektur nach Ferstl und Sinz beschreibt nun gerade die Traces zwischen diesen drei Ebenen aus betriebswirtschaftlich relevantem Blickwinkel.

Je nachdem welche der drei Komponenten man näher betrachten möchte, kann man von den Interaktionen zwischen den anderen Systemen abstrahieren, indem man sie als eine zusammenhängende Komponente betrachtet. Im Rahmen dieser Arbeit stehen insbesondere die Interaktionen zwischen dem Anwendungssystem und seiner Umwelt – diese soll im Weiteren mit ENV_{AWS} bezeichnet werden – im Vordergrund. Wie die Aufbauorganisation sowie die Maschinen und die Anlagen intern strukturiert sind, ist nicht von Interesse, und wie sie in Bezug auf ein Anwendungssystem interagieren sollen, wird i.d.R. in dem Soll-Konzept vorgegeben. Das heißt, dass die internen Aktionen der Aufbauorganisation und der Maschinen mit internen τ-Aktionen in Prozesskalkülen wie CCS (siehe [Milner 1989]) vergleichbar, also nach außen hin nicht sichtbar sind.

10.2.1 Exportierte Schnittstellen zur Umwelt

In einem herkömmlichen Anwendungssystem sind die möglichen Interaktionen zur Umwelt durch System- und Benutzungsschnittstellen beschrieben. In SCDL wurde bislang keine

Unterscheidung zwischen den Schnittstellen zu Komponenten und den Benutzungs- und Systemschnittstellen getroffen. Deshalb sollen nun zuerst die Prozeduren, die von Benutzerseite und anderen Systemen (z.B. Maschinen) zugänglich sein sollen, explizit als solche gekennzeichnet werden. Eine andere Möglichkeit bestände darin, Benutzungs- und Maschinenkomponenten explizit zu definieren, die dann als Umwelt auf das Anwendungssystem zugreifen. Es soll der erste Weg verfolgt werden, da der zweite die Menge der Komponentenbeschreibungen unnötigerweise erhöhen und zudem erfordern würde, dass alle Anforderungen in SCDL beschrieben werden.

Da es denkbar ist, dass in einer Konfiguration die Schnittstelle einer Komponente direkt durch die Umgebung (ENV_{AWS}), aber in einer anderen Konfiguration überhaupt nicht oder nur indirekt, d.h. über eine weitere Komponente, aufgerufen werden darf und folglich die Schnittstellen, die durch die Umwelt genutzt werden dürfen, von Konfiguration zu Konfiguration differieren können, müssen diese Informationen wie die Parametrierungsinformationen von Framework-Komponenten ebenfalls zusätzlich zu den Komponenten verwaltet werden können.

Die aufrufbaren Schnittstellen zur Umwelt werden als Menge von Signaturen ($Interface_{ENV}$) angegeben. Zur Vereinfachung der Schreibweise können auch ganze Komponenten und sogar Komponentenhierarchien – zur Erinnerung: die Komponentennamen stellen wie die Paketnamen in Java eine Hierarchie dar – angegeben werden. Bei der Angabe von Komponenten oder Hierarchien von Komponenten sind alle Signaturen des Elementes aufrufbar. Da Komponenten, Klassen und Prozeduren eindeutig durch einen Pfadausdruck (PATH, siehe Abschnitt 7.2) identifiziert werden können, hat $Interface_{ENV}$ folgende Gestalt:

$Interface_{ENV}$ = { <PATH1>, ..., <PATHn>, <SIG1>, ..., <SIGm> }

<PATHi> steht entweder für eine Menge von Komponenten, für eine einzelne Komponente (mit allen direkten und indirekten Prozeduren) oder für eine einzelne Klasse (mit allen ihren Methoden). <SIGj> (siehe Abschnitt 8.2.4) steht für eine einzelne Prozedur (oder Methode).

Nach welchen Kriterien der Anwendungsarchitekt die Schnittstellenmengen definiert, hängt letztlich von dem Anwendungsbereich des Anwendungssoftwareproduktes und seinen Entwurfsentscheidungen ab. Generell lässt sich sagen, dass in der Umgebung ENV_{AWS} eines Anwendungssystems die erlaubten Aufrufe unterschiedlich detailliert vorgegeben sind. Aus Sicht der Maschinen der Umgebung ENV_{AWS} werden Aufrufe in der Regel durch Prozesssteuerungssysteme initiiert, die betriebliche Anwendungssysteme z.B. über den Status von Fertigungsprozessen informieren können. Aus Sicht der Organisation werden u.a. Organisationshandbücher definiert, die der Nutzung (inkl. Aufruf) von Anwendungssystemen aus Sicht der Mitarbeiter dienen. Ein Anwendungsarchitekt kann versuchen eine Obermenge,

der in einer Branche benötigten Funktionalität über entsprechende Schnittstellen anzubieten, die dann zur Konfigurationszeit ausgewählt werden können.

10.2.2 Importierte Schnittstellen der Umwelt

Da auch aktive Informationssysteme zum Einsatz kommen und somit nicht nur der Benutzer ein Anwendungssystem, sondern auch ein Anwendungssystem einen Benutzer von sich aus informieren („aufrufen“) kann, spielen auch die Schnittstellen der „Komponente“ Benutzer als Teilkomponente der Aufbauorganisation eine wichtige Rolle. Da die Fähigkeiten und Anforderungen eines Benutzers (sowie der Maschinen und Anlagen) auch im Rahmen eines Soll-Konzeptes nicht erschöpfend beschrieben werden können, wird in dieser Arbeit so verfahren, dass alle Slots für Parameter-Komponenten auch durch die Komponente $\mathtt{ENV}_{\mathtt{AWS}}$ besetzt werden können, ohne das mögliche Verhalten dieser Komponente explizit angeben zu müssen. Natürlich ist auch die Entscheidung, wann ein Slot durch eine konkrete Softwarekomponente besetzt werden soll und wann nicht, konfigurationsabhängig. In stark automatisierten Unternehmen werden dann möglichst viele Slots durch Software-Komponenten besetzt. In anderen Unternehmen wird oft noch der Anwender zu Rate gezogen.

Es wird davon ausgegangen, dass die Komponente $\mathtt{ENV}_{\mathtt{AWS}}$ alle Kontrakte der Slots, in die sie eingesetzt wird, standardmäßig erfüllt. Das bedeutet, dass z.B. ein Framework, der Slots für Komponenten zur Übermittlung von Bestellaufträgen besitzt, diese Aufträge entweder von einer „richtigen“ Komponente oder von der Umweltkomponente $\mathtt{ENV}_{\mathtt{AWS}}$ zugestellt bekommen kann. Beide Komponenten wären im Sinne der Konfiguration Alternativen. Der Anwendungsarchitekt (siehe 1.2) muss entscheiden, welche Interaktionen für den Benutzer (und die Maschinen) zumutbar bzw. realisierbar sind und welche nicht[1].

Offensichtlich benötigt der Benutzer zur Einstellung von Bestellaufträgen eine Benutzungsschnittstelle (GUI). Da diese Arbeit sich jedoch ausschließlich auf die Anwendungsebene bezieht, wird von diesem Problem abstrahiert.

---

[1] Indem man die Kontrakte, die die Komponente $\mathtt{ENV}_{\mathtt{AWS}}$ innerhalb einer konkreten Konfiguration tatsächlich einhalten muss, bestimmten Organisationseinheiten oder Stellen des Unternehmens zuteilt, kann man erste Aufgabenprofile der Mitarbeiter in Form von Rollen definieren. Auf diese Möglichkeiten wird nicht weiter eingegangen. Dieser Aspekt scheint aber insoweit interessant zu sein, als dadurch z.B. während der Erstellung von Geschäftsprozessen Informationen für die Aufbauorganisation abgeleitet werden können.

10.2.3 Geschäftsprozesse sind fachliche Traces

Betriebliche Anwendungssysteme zeichnen sich dadurch aus, dass Benutzer aus einer Menge unterschiedlicher betrieblicher Transaktionen auswählen können. Eine *betriebliche Transaktion* ist eine Interaktion zwischen der „Komponente“ Benutzer und einer Softwarekomponente des Anwendungssystems.[1] Eine solche Interaktion stößt eine Reihe von weiteren Interaktionen zwischen verschiedenen Softwarekomponenten an, bis ein Ergebnis an den Benutzer zurückgeliefert wird.

Durch moderne Technologien wie z.B. Wiedervorlage- oder Workflow-Managementsysteme (siehe [Jablonski und Bussler 1996]) gelangen zunehmend aktive, steuernde Aspekte in die betrieblichen Anwendungssysteme. Nun muss der Benutzer nicht mehr das System ständig nach neuen Informationen abfragen (Pull-Modell, siehe Abschnitt 3.3.3), sondern wird beim Auftreten interessanter Sachverhalte direkt durch das System informiert und kann auch zur Durchführung bestimmter Handlungen aufgefordert werden. Solche Aufforderungen werden oft als *Work-Items* (*Arbeitsposten*) bezeichnet.

Betriebliche Transaktionen sind also Aufforderungen des Anwenders an das Anwendungssystem, etwas durchzuführen, und Work-Items sind Aufforderungen, die aus Prozessen des Anwendungssystems resultieren und den Benutzer dazu animieren sollen, seinerseits bestimmte betriebliche Interaktionen mit dem System durchzuführen. Beide Konzepte lassen sich auf das Konzept „Schnittstelle“ reduzieren. Betriebliche Transaktionen sind besondere Schnittstellen von Komponenten, die auch oder ausschließlich von Benutzern aufgerufen werden dürfen, Work-Items sind hingegen „Aufrufe“ von Schnittstellen der „Komponente“ Benutzer.

Geschäftsprozesse lassen sich dann – zumindest soweit sie mit einem betrieblichen Anwendungssystem in Zusammenhang stehen – als Folgen von Schnittstellenaufrufen (Interaktionen) zwischen Komponenten und Benutzern verstehen, die der Verfolgung eines betrieblichen Ziels dienen[2].

---

[1] Die insbesondere von betrieblichen Transaktionen – bzw. den dazu benötigten Datenbanktransaktionen – geforderten ACID-Eigenschaften (Atomicity, Consistency, Isolation, Durability) spielen in dieser Arbeit keine Rolle. Es wird davon ausgegangen, dass entsprechende Dienste durch Komponenten-Frameworks zur Verfügung gestellt werden. Im Rahmen der Konfiguration soll der Konfigurierer von diesen Problemen abstrahieren – ähnlich wie ein Berater sich auch nicht für die Persistenzmechanismen des R/3-Systems interessiert.

[2] Die Unterscheidung von Aufruf einer Transaktion einer Komponente und der Benachrichtigung eines Benutzers wird z.B. in der EPK-Sprache nicht gemacht.

Aus dieser abstrakten Sicht kann man Geschäftsprozesse nun mit Traces vergleichen wie sie auch in der Theoretischen Informatik bekannt sind und in Abschnitt 7.4 vorgestellt wurden. *Traces* sind als endliche Sequenzen von Kommunikationen (Interaktionen) zwischen Prozessen definiert. Im Sinne dieser Arbeit werden Traces und damit (anwendungsbezogene) Geschäftsprozesse als Sequenzen von Interaktionen zwischen Softwarekomponenten und Benutzern verstanden. In Bezug auf die Beschreibungssprache SCDL besteht eine Interaktion aus einem Aufruf einer Komponentenschnittstelle durch eine andere Komponente. Die empfangende Komponente erhält dazu benötigte Informationen und liefert nach Abarbeitung einer Prozedur oder Methode veränderte oder neue Informationen an die aufrufende Komponente zurück. In der Zwischenzeit können weitere Interaktionen zu anderen Komponenten stattfinden. Die Folgen von Interaktionen können nacheinander oder parallel ablaufen.

Zur Veranschaulichung lässt sich der in Abschnitt 7.4 präsentierte und in Abschnitt 8.2.5 zu einem Geschäftsprozess konkretisierte Trace in EPK-ähnlicher Form darstellen. In Abbildung 10.3 sieht man den Trace als Geschäftsprozess in EPK-Notation. Der Trace ist sequentiell, da alle Aufrufe von Prozeduren aus einem sequentiellen Prozess (p3, vgl. Abschnitt 7.4) heraus stattfinden. Dieser Prozess wird hier als EPK veranschaulicht. Die anderen Prozesse (p1, p2, p4 und p5) laufen parallel, sind aber nicht aktiv und können deshalb nicht explizit als EPK dargestellt werden.

Interaktionen werden als EPK-Funktionen und Zustände als EPK-Ereignisse beschrieben. Die Zustände werden durch Konnektoren zu Situationen zusammengefasst. Während in Geschäftsprozessen typischerweise Organisationseinheiten für die Durchführung von Aufgaben zuständig sind, werden hier links vom Funktionssymbol, die aufrufende (oder: ausführende) Komponente und rechts vom Funktionssymbol die empfangende Komponente (oder auch: das empfangende Objekt) sowie beteiligte Objekte (= aktuelle Parameter) aufgeführt. Das Entitätstypen-Symbol wird hier für Objektinstanzen zweckentfremdet. Damit wird dem Manko der EPK begegnet, dass sie kein Sprachelemente zur Bezeichnung von Instanzen (z.B. Variablen) besitzt.

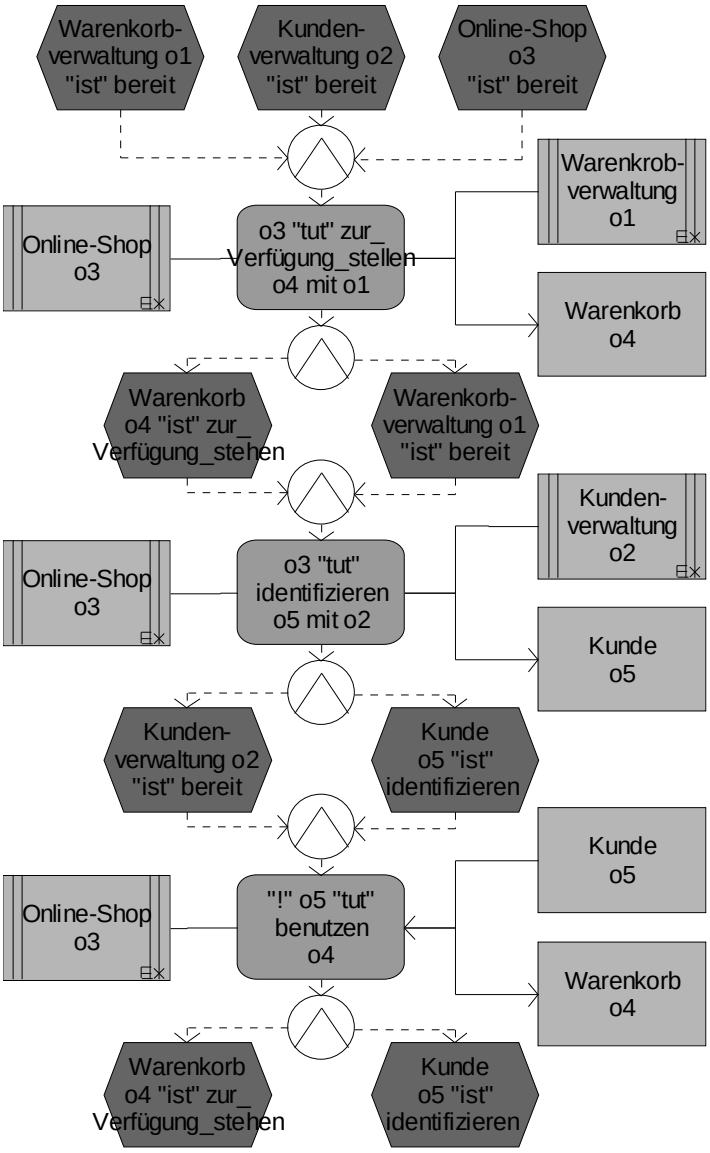


Abbildung 10.3: Trace in EPK-Darstellung

10.3 Ableitung von Geschäftsprozessen

Die Umgebung ENV_{AWS} des Komponentenmodells ist, wie in den vorhergehenden Abschnitten erläutert wurde, eine Komponente, die den bzw. die Interaktionspartner des Anwendungssystems darstellt, aber aufgrund seiner Komplexität nicht näher beschrieben wird. Um im Konfigurationsprozess die Anforderungen dieser Komponente – und damit die Anforderungen an das zukünftige Anwendungssystem – zu berücksichtigen, wird sie im konkreten Konfigurationsprozess durch den Konfigurierer „simuliert“. Der Konfigurierer ist also für die Umsetzung der Anforderungen des Anwenders und damit für die Umsetzung des Soll-Konzeptes zuständig. Er darf bei der Simulation der Geschäftsprozesse nur Prozesse erzeugen, die auch tatsächlich im Soll-Konzept gefordert werden.

In Prozessterm-Schreibweise kann man diesen Sachverhalt folgendermaßen formulieren:

$$ENV_{AWS} = Konfigurierer \mid Soll\text{-}Konzept$$

Beim Konfigurationsvorgang kommuniziert ENV_{AWS} mit den Komponenten des Anwendungssystems, legt aber dabei nur ein von dem Anwendungssystem erlaubtes Verhalten an den Tag. Können nun bestimmte geforderte Geschäftsprozesse nicht erzeugt werden – z.B. weil die Funktionalität der Komponenten bestimmte Interaktionen nicht zulässt –, dann deckt das Anwendungssoftwareprodukt die Anforderungen des Anwenders nicht ab und muss offensichtlich erweitert werden, bzw. die Anforderungen müssen revidiert werden. Anderenfalls können zumindest einzelne Geschäftsprozesse als realisierbar nachgewiesen werden. Dieses Vorgehen soll im nächsten Abschnitt weiter verfolgt werden.

10.4 Algorithmus zur prozessorientierten Konfiguration

Die prozessorientierte Konfiguration vollzieht sich grob durch die Nutzung von vier Funktionen:

- Erstellen einer initialen Konfiguration,
- Konstruktion eines Geschäftsprozesses (und der damit verbundenen Konfiguration des Anwendungssoftwareproduktes),
- Zurücknahme eines Geschäftsprozesses (und der damit verbundenen Rücknahme der vorher getroffenen Konfigurationsentscheidungen) und
- Prüfen der Konfiguration.

Die Zusammenhänge der verschiedenen Informationen, die bei der Konfiguration erzeugt und genutzt werden, werden in Abbildung 10.4 veranschaulicht. Alle Konfigurationsentscheidungen werden im Rahmen der Simulation von Interaktionen getroffen, die zusammen einen Geschäftsprozess ergeben. Eine Konfigurationsentscheidung gibt für eine abstrakte Komponente eine ausgewählte (konkrete) Komponente an. Unter einer abstrakten bzw. gebundenen Konfiguration verbirgt sich jeweils ein Tupel, das aus dem eigentlichem Anwendungssoftwareprodukt sowie den während des Konfigurationsprozesses abgeleiteten Geschäftsprozessen und den dabei angefallenen Konfigurationsentscheidungen besteht. Die gebundene Konfiguration unterscheidet sich von der abstrakten ausschließlich darin, dass im Rahmen des Konfigurationsprozesses keine weiteren Entscheidungen mehr getroffen werden müssen, um das Anwendungssoftwareprodukt einsetzen zu können. Ob jedoch weitere Konfigurationsentscheidungen zu treffen sind, um den Anforderungen des Anwenders gerecht zu werden, kann nur durch den Konfigurierer und nicht automatisch durch das System erkannt werden.

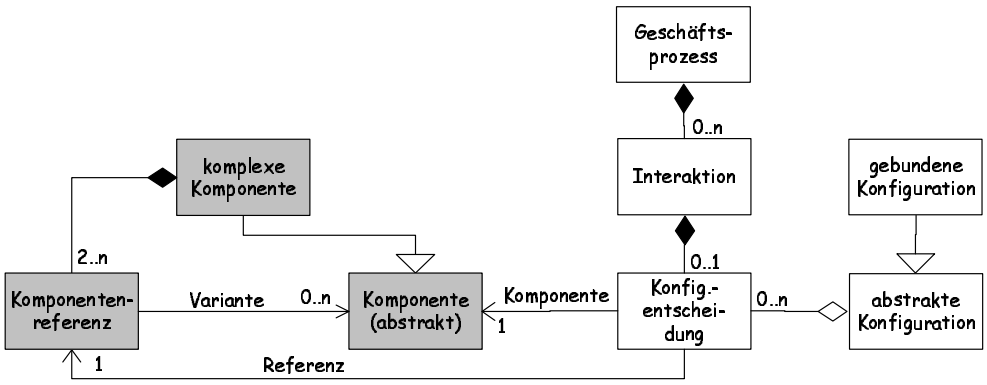


Abbildung 10.4: Konzepte der Konfiguration

Um eine Konfiguration (siehe Abbildung 10.5) durchzuführen, wird zunächst eine erste, initiale (abstrakte) Konfiguration (siehe Abschnitt 10.4.1) erzeugt. Danach werden Geschäftsprozesse abgeleitet (Abschnitt 10.4.2), wodurch die Konfiguration des Anwendungssoftwareprodukts zunehmend konkreter wird, bis letztlich eine gebundene Konfiguration („Prüfen der Konfiguration“, siehe Abschnitt 10.4.4) erreicht wird. Da der Anwender unter Umständen bestimmte Anforderungen revidiert, muss auch der Konfigurierer in der Lage sein, ausgewählte Geschäftsprozesse zu ändern oder vollständig zurückzunehmen. Hier wird nur die Zurücknahme von Geschäftsprozessen (siehe Abschnitt 10.4.3) betrachtet, da die Änderung sich durch Zurücknahme und erneute Konstruktion simulieren lässt. Durch die Zurücknahme von Geschäftsprozessen werden Konfigurationsentscheidungen revidiert, was dazu führt, dass eine Konfiguration wieder abstrakter wird.

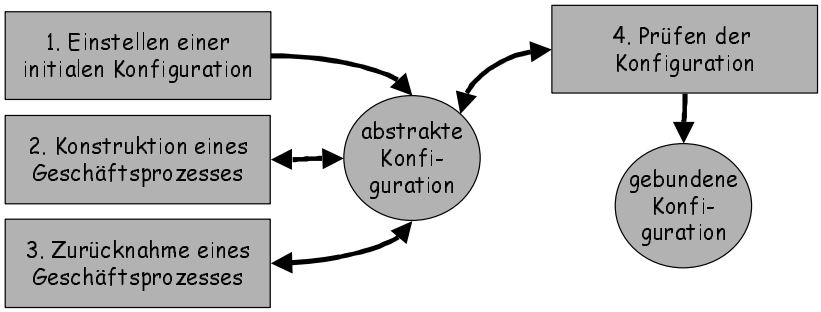


Abbildung 10.5: Prozessorientierte Konfiguration (Grobsicht)

10.4.1 Einstellen einer initialen Konfiguration

Zu Beginn des Konfigurationsprozesses wird eine initiale Konfiguration erzeugt. Eine solche Konfiguration wird durch den Operator `newConfiguration` erzeugt, der als Eingabe eine Beschreibung eines Anwendungssoftwareprodukts auf Basis von SCDL und CCL erhält. Neben dieser Konfiguration wird eine Menge von Geschäftsprozessen verwaltet. An jedem Geschäftsprozess werden die Auswahlentscheidungen protokolliert (siehe Abschnitt 10.4.2), die im Rahmen der Ableitung des Geschäftsprozesses getätigt wurden.

10.4.2 Konstruktion eines Geschäftsprozesses

Im nächsten Schritt stehen die „Konstruktion der Geschäftsprozesse“ und die damit verbundenen Berechnungen der Auswirkungen auf die Konfiguration an. Dazu wird im Folgenden ein Algorithmus (siehe Abbildung 10.6) angegeben, der als Eingabe eine abstrakte Konfiguration in Form einer Beschreibung B eines Anwendungssoftwareprodukts auf Basis von SCDL und CCL bekommt. Als Ergebnis wird ein Geschäftsprozess G und eine damit zusammenhängende, gebundene Konfiguration K des Anwendungssoftwareprodukts für ein konkretes Anwendungssystem zurückgegeben.

Eingabe: Anwendungssoftwareprodukt `B`.

Ausgabe: Geschäftsprozess `G`, Konfiguration `K`.

1. `G = [ ], K=K‘;`

2. Berechnung der Menge `I` aller möglichen Interaktionen von `B`. Zu diesen Interaktionen gehören alle für die Umgebung ENV_{AWS} zugelassenen Schnittstellenaufrufe des Anwendungssoftwareprodukts.

3. Die Menge `I` der Interaktionen wird unter Berücksichtigung der normsprachlichen Annotationen dem Konfigurierer aufgelistet.

4. Auswahl einer Interaktion `i ∈ I` (bzw. ihrer Annotation `j`) durch den Konfigurierer. `G = append(G‘, j)`. Wird durch die Auswahl auch eine Komponente `c` vom Typ `C` für einen Slot `s` eines Frameworks `f` vom Typ `F` bestimmt, so wird die Konfiguration folgendermaßen erweitert:

```
K= addComponent(K‘);                    /* Komponente c erzeugen */
K= spezializeComponent(K‘, c, C);       /* Komponente c spezial.*/
K= addReference(K‘, f, c, s);           /* Komponente c einhängen */
```

5. Berechne alle möglichen Folgeinteraktionen I der Interaktion i.

6. Wenn der Geschäftsprozess „fertig“ konstruiert ist, breche ab, sonst gehe zu 3.

Abbildung 10.6: Algorithmus zur Konstruktion von Geschäftsprozessen

zu 1. Das Ergebnis dieses Algorithmus ist ein Geschäftsprozess – genauer: die abgeleitete Beschreibung eines Geschäftsprozesses. Eine Geschäftsprozess ist eine Folge (Liste) von Interaktionen, die durch den Algorithmus abgeleitet wird.

zu 2. Da im Rahmen der Konfiguration die „richtigen“ Softwarekomponenten ausgewählt werden müssen, ergibt sich die Menge der alternativen Interaktionen zum einen aus der

Ablauflogik der schon ausgewählten Komponenten und zum anderen aus den Interaktionen der alternativ in Slots einzusetzenden Komponenten. Sobald für eine Interaktion eine Prozedur einer solchen Komponente ausgewählt wurde, wird damit auch die Auswahl der Komponente bestimmt (siehe Punkt 5).

zu 3. Alle Interaktionen werden, bevor sie dem Konfigurierer zur Auswahl vorgelegt werden, entsprechend einer vorgegebenen Terminologie normsprachlich aufbereitet („interpretiert“).

zu 4. Der Benutzer wählt nun eine Interaktion aus, die an den bisher abgeleiteten Geschäftsprozess angehängt (append) wird. Die Konfiguration wird für den Fall, dass durch die Interaktionsauswahl auch die Auswahl einer Komponente getroffen wurde, auf Basis der Konfigurationsoperatoren (siehe Abschnitt 9.3.6) erweitert. Die dadurch eingestellten assertionalen Axiome werden von dem Konfigurationssystem verarbeitet und propagiert.

zu 5. Zur weiteren Ableitung werden alle Interaktionen berechnet, die der zuvor ausgewählten Interaktion im Sinne eines Geschäftsprozesses folgen können. Da die Koordination der einzelnen Arbeitsschritte eines Geschäftsprozesses innerhalb eines Anwendungssoftwareprodukts nicht generell automatisiert durchgeführt wird, können prinzipiell alle durch die Umgebung (ENV_{AWS}) initiierbaren Interaktionen ausgewählt werden[1]. Ob sie in einem inhaltlichen Kontext stehen, kann nur der Konfigurierer entscheiden.

zu 6. Wann ein Geschäftsprozess „fertig“ ist, kann der Konfigurierer bestimmen.

10.4.3 Zurücknahme eines Geschäftsprozesses

Alle Entscheidungen, die im Rahmen der Konfiguration getroffen wurden, dürfen durch den Konfigurierer rückgängig gemacht werden. Dadurch wird es ihm ermöglicht, relativ einfach verschiedene Konfigurationen mit unterschiedlichen Geschäftsprozessen durchzuspielen. Da Konfigurationsentscheidungen unmittelbare Auswirkungen auf die möglichen Geschäftsprozesse haben, muss insbesondere auf die Wahrung der Konsistenz zwischen Geschäftsprozessen und Konfigurationsentscheidungen geachtet werden. Denn falls eine Konfigurationsentscheidung rückgängig gemacht werden soll, müssen auch die Konsequenzen für den Geschäftsprozess beachtet werden. Der hier verfolgte Ansatz zur Konsistenzwahrung ist relativ einfach. Er erlaubt nur die Rücknahme aller Konfigurationsentscheidungen eines Geschäftsprozesses. Feinere Mechanismen sind natürlich notwendig, werden aber in dieser Arbeit nicht weiter verfolgt.

Die Informationen darüber, welche Entscheidungen durch die Konstruktion welches Geschäftsprozesses entstanden sind, werden während der Konfiguration zu einem

[1] Ausnahmen würden Anwendungssoftwareprodukte mit vorkonfigurierten Workflows darstellen.

Geschäftsprozess (siehe Abschnitt 10.4.2 und Abbildung 10.4) gespeichert. Um diese Entscheidungen wieder rückgängig zu machen, müssen entsprechende Operatoren aus Abschnitt 9.3.6 ausgewählt werden. Für jede Komponente, die im Rahmen der Konstruktion ausgewählt wurde, wurden die folgenden drei Operatorenaufrufe (siehe Abbildung 10.5) durchgeführt:

```
K= addComponent(K‘);
K= spezializeComponent(K‘, c, C);
K= addReference(K‘, f, c, s);
```

Zur Zurücknahme müssen nun entsprechend die drei inversen Operatoren aufgerufen werden:

```
K= delReference(K‘, f, c, s);
K= generalizeComponent(K‘, c);
K= delComponent(K‘, c);
```

Etwaige Propagierungen, die von dem Konfigurationssystem aufgrund von verschiedenen Constraints vorgenommen wurden, werden bei diesem Vorgang von einem zugrunde liegenden System für Beschreibungslogiken automatisch zurückgenommen.

10.4.4 Prüfen der Konfiguration

Die Prüfung, ob eine Konfiguration gebunden ist oder nicht, kann durch die Operation `isBoundConfiguration` vorgenommen werden. Für die weitere Anwendung, z.B. um ein reales Anwendungssoftwareprodukt – und nicht nur seine Beschreibung – nun endgültig zu konfigurieren, ist es erforderlich, die gebundene Konfiguration „auszulesen“. Dieser Vorgang ist relativ einfach, da es sich bei einer gebundenen Konfiguration um einen Baum handelt, dessen Knoten in beliebiger Reihenfolge traversiert werden kann.

10.5 Fazit

Mit Abschluss dieses Kapitels schließt nun auch der Teil über die Konzeption der geschäftsprozessorientierten Konfiguration komponentenbasierter Anwendungssysteme. Mit der Definition

- einer Komponentenbeschreibungssprache (SCDL),
- einer Konfigurationssprache (CCL) zur Beschreibung von Anwendungssoftwareprodukten und
- einer Methode zur „geschäftsprozessorientierten Konfiguration“ dieser Produkte zu einsetzbaren Anwendungssystemen

sind alle Grundlagen für die Realisierung eines entsprechenden Konfigurationssystems geschaffen worden.

Teil III Implementierung von PROSECCO

11 Architektur

In diesem Kapitel wird die Architektur des Softwarewerkzeugs PROSECCO beschrieben, das den Konfigurierer bei der prozessorientierten Konfiguration eines Anwendungssoftwareprodukts (siehe Kapitel 10) unterstützen soll. Das Akronym PROSECCO steht für „Process descriptions for the software engineer - consultant - customer - supply chain“ und soll die Forderung nach einer anwendungsnahen Unterstützung der „Softwarekomponenten-Zulieferkette“ durch verhaltensbezogene Beschreibungen unterstreichen (siehe auch [Ritter 1998b]).

Die hier entwickelte Architektur dient in erster Linie einer zusammenhängenden Darstellung der im Rahmen dieser Arbeit entwickelten Konzepte und dann als Grundlage für eine Implementierung eines prozessorientierten Konfigurationssystems. Bislang wurden nur ausgewählte Teile des Systems implementiert, die in Kapitel 12 vorgestellt werden. Die Implementierung diente eher einer Überprüfung der Machbarkeit („proof of concepts“) eines solchen Konfigurationssystems, als der Entwicklung eines in der Praxis tatsächlich einsetzbaren Werkzeugs.

11.1 Architekturübersicht

Das zu entwerfende Werkzeug wurde in Hinblick auf eine klassische Drei-Ebenen-Architektur konzipiert. Die oberste Ebene (*Benutzungsebene*) stellt die Schnittstelle zum Benutzer (Konfigurierer) dar. Die mittlere Ebene (*Anwendungsebene*) dient der Anwendungslogik. In dieser Ebene werden die Komponentenbeschreibungen und Konfigurationsinformationen zur Laufzeit des Konfigurationssystems bearbeitet. Die untere Ebene ist die *Persistenzebene*, in der die Beschreibungen der Komponenten bzw. der Anwendungssoftwareprodukte persistent verwaltet werden. Das Kernstück dieser Arbeit stellt sicherlich die Anwendungsebene dar. Sie erhält von der Persistenzebene die SCDL-Komponentenbeschreibungen, die CCL-Konfigurationsbeschreibungen und die normsprachlichen Annotationen. Alle drei Beschreibungen zusammen beschreiben ein Anwendungssoftwareprodukt. Außerdem können abstrakte Konfigurationen als Zwischenstände und gebundene Konfigurationen dauerhaft gespeichert werden und ggf. zu späteren Zeitpunkten wieder geladen und weiter verarbeitet werden.

Auf der Ebene des Benutzers bietet das Konfigurationssystem während des Konfigurationsvorgangs alternative, durch Annotationen interpretierte Interaktionen an, die im Rahmen des Konfigurationsprozesses ausgewählt werden müssen. Eine ausgewählte Interaktion wird zusammen mit den dadurch evtl. ermittelten Konfigurationsentscheidungen im Zusammenhang mit einer Geschäftsprozessbeschreibung gespeichert. Um Konfigurationsentscheidungen rückgängig machen zu können, kann der Benutzer einen Geschäftsprozess auswählen. Alle Konfigurationsentscheidungen, die im Rahmen der

Ableitung des Geschäftsprozesses getroffen wurden, können dann in einem Schritt rückgängig gemacht werden. Dazu werden die „inversen Konfigurationsentscheidungen“, also die Umkehrungen der ursprünglich getroffenen Entscheidungen berechnet.

Die Abbildung 11.1 zeigt die Grobarchitektur in Form der graphischen Darwin-Notation[1], die als Architekturbeschreibungssprache in Abschnitt 4.4 vorgestellt wurde. Die wichtigsten Komponenten dieser Architektur werden in diesem und im nächsten Kapitel weiter zerlegt und dabei genauer beschrieben. Hier sind zunächst nur die Ebenen der Drei-Ebenen-Architektur und die Schnittstellen zwischen diesen Ebenen zu sehen.

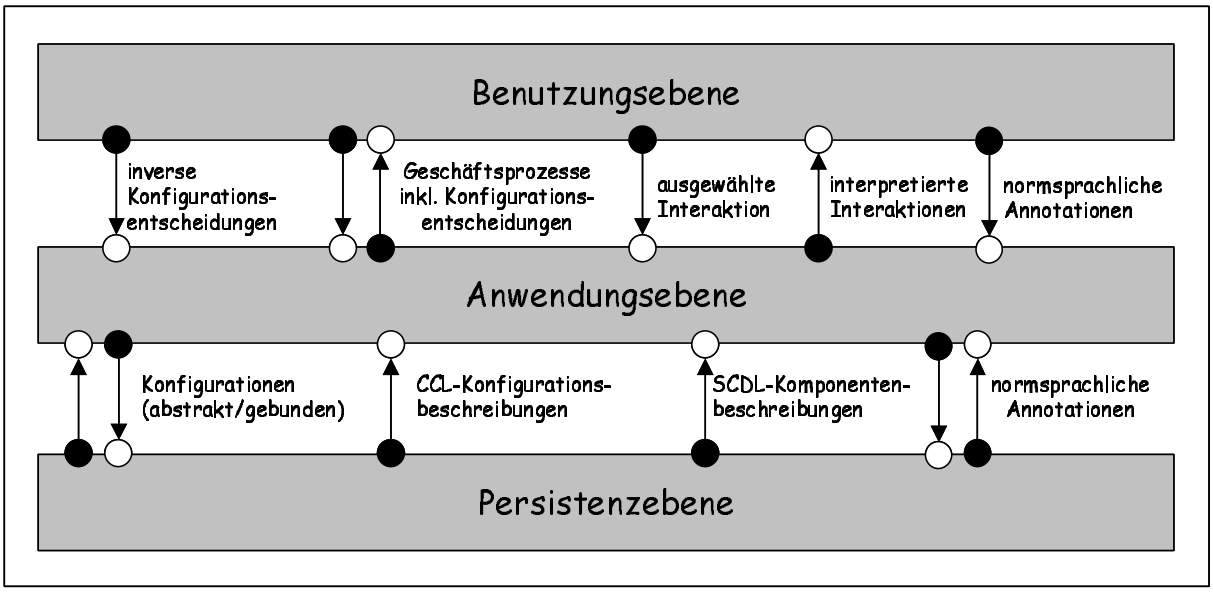


Abbildung 11.1: Grobarchitektur

11.2 Anwendungsebene

In der Anwendungsebene (siehe Abbildung 11.2) des Konfigurationssystem sind drei Hauptmodule definiert,

- der *Simulator*, der im Wesentlichen eine Laufzeitumgebung für die SCDL-Beschreibungen der Komponenten eines Anwendungssoftwareprodukts darstellt,

[1] Weiße Rechtecke definieren komplexe und graue Rechtecke definieren deren konstituierende Komponenten. Eine Komponenten hat ausgehende (schwarze Punkte) und eingehende Schnittstellen (weiße Punkte). Sowohl ein- als auch ausgehende Schnittstellen einer konstituierenden Schnittstelle können an die umgebende komplexe Komponente exportiert werden.

- der *Konfigurator*, der die Konfiguration eines Anwendungssoftwareprodukts verwaltet und manipuliert, die auf der Grundlage von CCL angegeben wird, und
- der *Fachübersetzer*, der für die inhaltliche Interpretation der Interaktionen sorgt, indem die formalen Interaktionen gemäß der zu einem Anwendungssoftwareprodukt definierten Annotationen normsprachlich übersetzt werden.

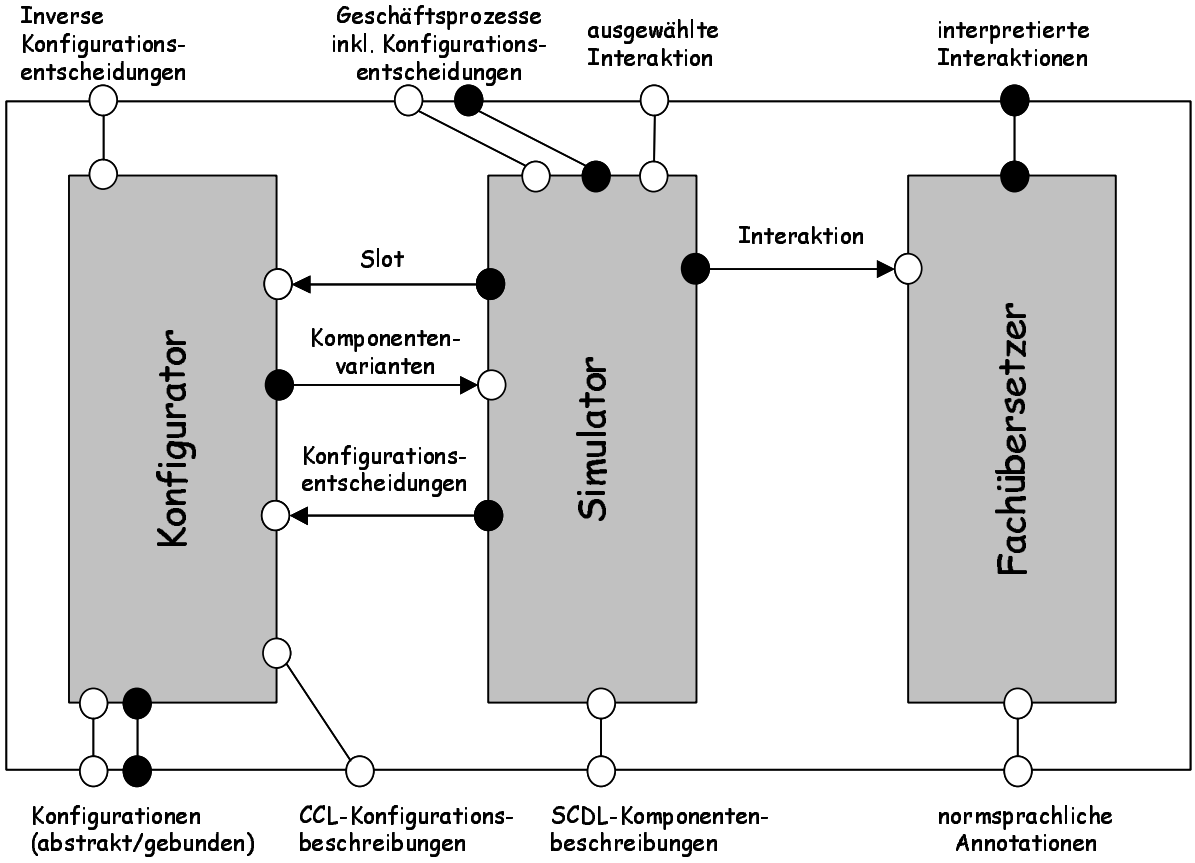


Abbildung 11.2: Anwendungsebene

11.2.1 Simulator

Der Simulator stellt eine Laufzeitumgebung für die SCDL-Beschreibungen eines Anwendungssoftwareprodukts dar. Dazu erhält der Simulator als Eingabe eine Menge von SCDL-Beschreibungen, in der die Komponenten dieses Produkts beschrieben sind. Er startet die Simulation, indem alle Komponenten initialisiert werden. Nach der Initialisierung beginnen die Komponenten miteinander zu kommunizieren. Alle Interaktionen zwischen diesen Komponenten, die nicht von einer Konfigurationsentscheidung abhängig sind, können direkt ausgeführt und dem Konfigurierer über die Benutzungsebene angezeigt werden. Da die SCDL-Interaktionen relativ schwer verständlich sind, werden sie dazu zuvor durch den

Fachübersetzer aufbereitet. Dieser Übersetzer (siehe Abschnitt 11.2.3) ist eine Umsetzung des in Kapitel 8 konzipierten Interpretationsansatzes.

Einige Komponenten aus der SCDL-Beschreibung sind Komponenten-Frameworks und besitzen folglich Slots, die noch nicht mit konkreten Komponenten „bestückt“ wurden und für die noch eine Komponente ausgewählt werden muss. Immer wenn der Simulator auf einen solchen noch nicht konfigurierten Slot stößt, ermittelt er über den Konfigurator die zu diesem Slot in der Konfiguration erlaubten Komponenten (Komponentenvarianten).

Aus den Komponentenvarianten werden nun die möglichen Interaktionen ermittelt, die bei einem Einsatz dieser Komponente als nächstes ausgeführt werden würden. Diese Interaktionen werden dem Konfigurierer (nach entsprechender Aufbereitung durch den Fachübersetzer) zur Auswahl angeboten.

Über die Benutzungsebene bekommt der Simulator nun die ausgewählte Interaktion mitgeteilt und kann darüber die ausgewählte Komponente ermitteln. Diese Konfigurationsentscheidung (= Auswahl einer Komponente) wird der Konfiguratorkomponente mitgeteilt.

Zusätzlich werden alle Interaktionen, die im Rahmen der Simulation ausgewählt werden, in einer Liste als Geschäftsprozess gespeichert, um dem Konfigurierer (und dem Anwender) zu verdeutlichen, in welchem Kontext die Interaktionen stattfinden können. Damit später Konfigurationsentscheidungen rückgängig gemacht werden können, die im Rahmen der Ableitung eines Prozesses getroffen wurden, werden die Konfigurationsentscheidungen zusammen mit dem Geschäftsprozess gespeichert.

11.2.2 Konfigurator

Der Konfigurator verwaltet zur Konfigurationszeit die Komponenten und ihre Konfigurationsalternativen. Dazu wird zuerst die CCL-Beschreibung eines Anwendungs-softwareprodukts in den Konfigurator geladen. Da im Regelfall noch keinerlei Konfigurationsentscheidungen getroffen wurden, wird aus der CCL-Beschreibung eine „abstrakte Konfiguration“ (siehe Abschnitt 9.1) berechnet, die den initialen Stand der Konfiguration widerspiegelt.

Der Konfigurator bekommt vom Simulator Anfragen in Bezug auf mögliche Komponentenvarianten. Dazu wird ihm ein Slot übergeben, für den er dann alle einsetzbaren Komponenten berechnet. Diese „Komponentenvarianten“ gibt der Konfigurator daraufhin an den Simulator zurück. Es müssen immer eine oder mehrere Komponenten zurückgegeben werden. In dem Fall, dass keine Komponente zurückgegeben wird, bedeutet dies, dass die gesamte Konfiguration inkonsistent ist.

Hat der Konfigurierer eine Interaktion ausgewählt (siehe Abschnitt 11.2.1) und ist es eine Interaktion einer auswählbaren Komponente, wird dadurch gleichzeitig die Wahl der

Komponente bestimmt. Diese Komponentenwahl wird dem Konfigurator in Form von Konfigurationsentscheidungen mitgeteilt. Daraufhin propagiert der Konfigurator diese Informationen innerhalb seiner (abstrakten) Konfiguration. Durch diese Propagierung entlang von Constraints (siehe Abschnitt 9.3) werden u.U. die zukünftigen Auswahlmöglichkeiten für andere Slots weiter eingeschränkt, wodurch die Konfiguration zunehmend konkreter („gebundener“) wird. Diese zunehmende Konkretisierung macht sich aber erst während der weiteren Simulation bemerkbar. Ist eine Konfiguration nicht mehr abstrakt, gibt es also keine Auswahlmöglichkeiten innerhalb einer Konfiguration, so wird die Konfiguration als „gebundene Konfiguration“ ausgegeben. Sowohl abstrakte als auch gebundene Konfigurationen können in der Persistenzebene dauerhaft gespeichert werden und zu einem späteren Zeitpunkt zur weiteren Arbeit erneut geladen werden.

Da es auch möglich sein soll, Konfigurationsentscheidungen rückgängig zu machen, können „inverse Konfigurationsentscheidungen“ in den Konfigurator aufgenommen werden, wodurch die momentan bearbeitete Konfiguration wieder abstrakter wird. Die inversen Konfigurationsentscheidungen können aus den ursprünglichen Entscheidungen (eines Geschäftsprozesses) ermittelt werden und dann dem Konfigurator zugeführt werden.

11.2.3 Fachübersetzer

Der Fachübersetzer dient der fachlichen Interpretation der formalen SCDL-Interaktionen und der daraus abgeleiteten Traces (Geschäftsprozesse). Dazu werden die Interaktionen (siehe Abschnitt 7.4) im Sinne ihrer normsprachlichen Annotation übersetzt und dem Benutzer als interpretierte Interaktionen präsentiert. Die Art und Weise der Präsentation kann direkt aus den Ausführungen aus Abschnitt 8.3 abgeleitet werden.

Für den Konfigurationsprozess ist es wünschenswert, dass die Terminologie des späteren Anwenders nicht schon vorher vollständig definiert werden muss, sondern während der Konfigurationsprozesses in das System eingebracht werden kann. Dazu können die in Abschnitt 8.3 definierten Abbildungen zunächst als partielle Funktionen verstanden werden, deren Definitionsbereich während des Konfigurationsprozess zunehmend erweitert wird. Dazu kann der Anwender des Systems (Konfigurierer) nach und nach die gewünschten Ausdrücke und Satzformen seines Anwendungsbereichs als normsprachliche Annotationen über die Benutzungsebene in das System einbringen.

11.3 Persistenzebene

In der in Abbildung 11.1 dargestellten Architektur werden, die Daten auf der Persistenzebene in vier verschiedenen Datentöpfen verwaltet:

- einem für SCDL-Komponentenbeschreibungen,
- einem für CCL-Konfigurationsbeschreibungen,

- einem für normsprachliche Annotationen und
- einem für abstrakte bzw. gebundene Konfigurationen.

Die SCDL- und die CCL-Beschreibungen werden von außen in das System eingestellt, dazu werden entsprechende Editoren – im einfachsten Fall einfache Texteditoren – benötigt, die die Beschreibungen in den Datentöpfen abspeichern. Die Annotationen und die Konfigurationen können im Rahmen des Konfigurationsprozesses in die jeweiligen Datentöpfe eingestellt werden.

Um eine konsistente und standardisierte Speicherung der Daten zu ermöglichen, sollten alle Strukturen in einer strukturierten und offenen Art und Weise zugänglich sein. Dafür würden sich z.B. relationale Datenbank anbieten. Systeme, die diese Arten von Entwicklungsdokumenten (Metadaten) aufnehmen und über standardisierte Schnittstellen zur Verfügung stellen, werden auch als *Repositorien* (siehe [Habermann und Leymann 1993]) bezeichnet.

Um die angesprochenen Daten in einem Repository verwalten zu können, müssen die Sprachkonstrukte aus den entsprechenden Grammatiken (siehe für SCDL, Annotationen und CCL jeweils die Kapitel 7, 8 und 9) bzw. die Daten des Konfigurators (abstrakte und gebundene Konfigurationen) auf Relationen abgebildet werden. Ein erster Ausgangspunkt können dafür die in Abbildung 7.1 und Abbildung 10.4 dargestellten Klassendiagramme sein, die die benötigten Entitätstypen in UML-Notation (siehe [Fowler und Scott 1998]) darstellen.

Die Umsetzung dieser Ideen wäre für ein vollwertiges Konfigurationssystem unumgänglich. Da es sich bei PROSECCO jedoch lediglich um die Entwicklung eines Prototypen handelt, werden diese Ideen in dieser Arbeit nicht weiter verfolgt. Die Daten werden statt dessen in Textdateien in ihren ursprünglichen Formaten, die letztlich durch die verwendeten Werkzeuge und Programmiersprachen (siehe Kapitel 12) vorgegeben sind, verwaltet.

11.4 Benutzungsebene

Die Benutzungsebene dient als Schnittstelle zwischen System und Benutzer (Konfigurierer). Der Benutzer kann im Rahmen einer Konfiguration Geschäftsprozessbeschreibungen erstellen und wieder löschen, wodurch eine Spezialisierung bzw. Generalisierung der Konfiguration erreicht wird. Grundsätzlich kann er beliebig viele Geschäftsprozesse ableiten. Die einzige Bedingung, die dabei eingehalten werden muss, ist, dass die dabei entstehende Konfiguration konsistent bleibt. Dies wird ihm durch das Konfigurationssystem angezeigt.

12 Implementierungsaspekte

In diesem Kapitel werden einige Implementierungsaspekte von PROSECCO vorgestellt. Dazu werden zum einen die technische Realisierung der Komponenten genauer beschrieben und zum anderen auf Auswahlentscheidungen in Bezug auf Programmiersprachen und Programmentwicklungsumgebungen eingegangen. Als Implementierungsplattform wurde eine Sun-Solaris-Umgebung gewählt.

12.1 Anwendungsebene

Die Sprachen SCDL und CCL, die das formale Fundament dieser Arbeit darstellen, wurden auf Sprachen (π-Kalkül und Beschreibungssprachen) zurückgeführt, die wiederum formal fundiert sind und für die außerdem geeignete Entwicklungsumgebungen zur Verfügung stehen. Die Verfügbarkeit einer Implementierungsplattform hat so auch Einfluss auf die Wahl des formalen Fundaments dieser Arbeit gehabt. Die Anwendungsebene stellt den Kern der Implementierung dar. In dieser Ebene sind der Simulator, der Konfigurator und der Fachübersetzer angesiedelt. Die Implementierung dieser drei Teilkomponenten wird im Folgenden vorgestellt.

12.1.1 Simulator

Der Simulator (siehe Abbildung 12.1) dient als interaktiver Interpreter für die SCDL-Beschreibungen. Er besteht aus vier Komponenten:

- dem SCDL2Pict-Compiler,
- dem Pict-Compiler,
- der Laufzeitsystem und
- der Geschäftsprozessverwaltung.

12.1.1.1 SCDL2Pict-Compiler

Der SCDL2Pict-Compiler übersetzt eine Datei mit einer Menge von SCDL-Komponentenbeschreibungen nach Pict. SCDL wurde bewusst auf Basis einer Übersetzersemantik nach Pict definiert, da für diese Sprache eine Programmentwicklungsumgebung zur Verfügung steht. Im Rahmen der Arbeit wurde Pict 4.0 eingesetzt (siehe [Pierce und Turner 1997]).

Der SCDL2Pict-Compiler wurde in der Programmiersprache PROLOG (siehe [Clocksin und Mellish 1987]) implementiert, da PROLOG aufgrund der regelbasierten Sprachstruktur () zur prototypischen Implementierung von Parsern und Übersetzern besonders gut geeignet ist. Die

in Abschnitt 7.3 angegebenen Übersetzungsregeln lassen sich fast direkt in Hornklauseln überführen. Nur auf die syntaktischen Besonderheiten von PROLOG muss dabei Rücksicht genommen werden.

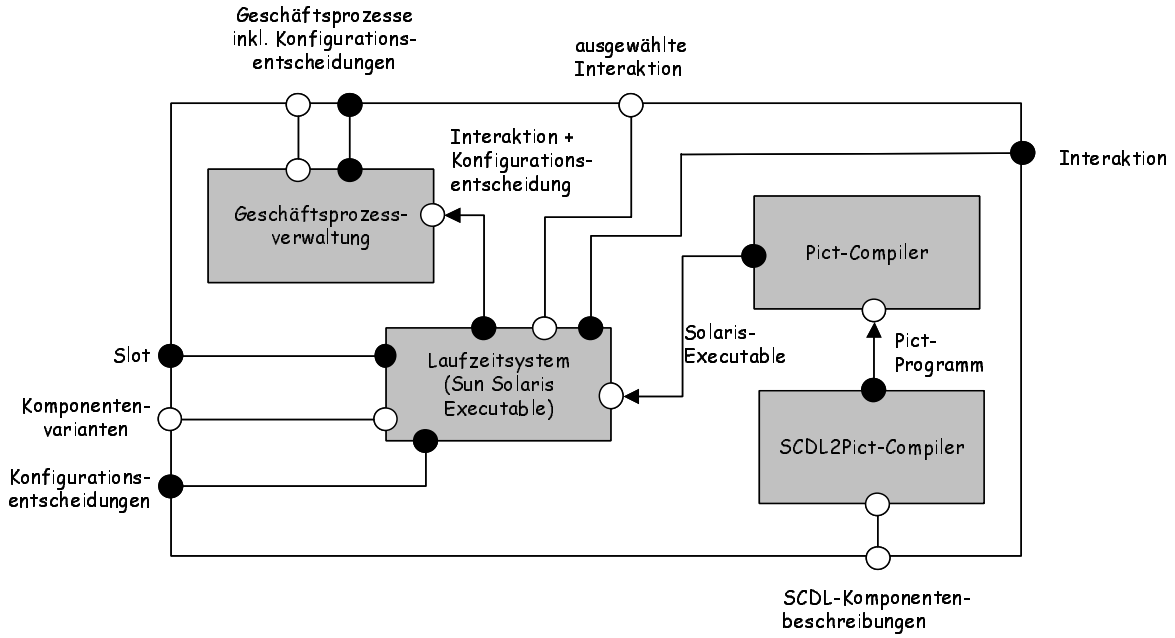


Abbildung 12.1: Simulator-Komponente

Ein kleines Beispiel einer implementierten Regel sieht man in Abbildung 12.2. Dies ist die Implementierung der Regel `transClass` aus Abschnitt 7.3.3. Jedoch handelt es sich hier um eine Regel, die nicht nur eine einzige, sondern eine ganze Liste von Klassen rekursiv abarbeitet und übersetzt (siehe hierzu auch die Ausführungen am Ende des Abschnitts 7.3.1).

```
transClasses(_N, _ID, [], _ENV).
transClasses(N, ID, [class(CID, _VARS, SIGS, BEHAV) | Rest], ENV):-
     catNames(ID,CID,NewID),
     transSignature(N, NewID, SIGS),
     transClassAgent(N, NewID, SIGS, BEHAV, ENV),
     transClasses(N, ID, Rest, ENV).
```

Abbildung 12.2: PROLOG-Regel des SCDL2Pict-Übersetzers

In Abbildung 12.3 wird das Resultat der SCDL2Pict-Übersetzung der Framework-Komponente `csFramework` (siehe Abschnitt 7.1) gezeigt.

Das Übersetzungsergebnis weicht in einigen Aspekten von der Konzeption aus Kapitel 7 ab. So besitzen alle SCDL-Komponenten und –Klassen zusätzliche Prozeduren. Mit der Prozedur `getOID` kann zur Laufzeit zu jeder Instanz ihre eindeutige Identifikationsnummer ermittelt werden. Ferner gibt es eine Prozedur `getType`, mit der der Bezeichner einer Komponente

bzw. Klasse ermittelt werden kann, zu der eine Instanz gehört. Jeder Prozeduraufruf besitzt neben seinen normalen SCDL-Parametern zwei Standardparameter (`pidStr` und `normalStr`), über die eine Nummer für den aktuellen Prozess (siehe Abschnitt 7.4) und ein String mit den Identifizierern der übergebenen aktuellen Parameter mitgegeben werden. Diese zusätzlichen Prozeduren und Parameter dienen ausschließlich der Erzeugung der Dokumentation der Interaktionen und damit der Kommunikation zwischen der Laufzeitumgebung und den restlichen Komponenten von PROSECCO (Fachübersetzer und Konfigurator). So wurde auch der Trace in Abschnitt 7.4 auf diese Weise automatisch erzeugt.

Es sei hier betont, dass alle vorgestellten Erweiterungen keinerlei Einfluss auf die Semantik von SCDL haben, sondern ausschließlich Implementierungszwecken dienen.

```
{- ********** COMPONENT: csFramework ************ -}
type csFramework = [
     getOID = /[!String]
     getType = /[!String]
]

def new_csFramework ( compId:String s:serverC c:clientC ): csFramework
=
     (val OID = (genOId)
     def getOID[s: !String] = s!OID
     def getType[s: !String ] = s!compId
     (
     {- translation of PROCEDURES -}
     {- translation of BEHAVIOR -}
          def initialized[pidStr: String normalStr:String ] =
          (new seq:^[]
          (pr![(+$ > "state(" pidStr "," (getType) "," (getOID)
                    ",initialized, [" "[]]),\n" ) (rchan seq)] |
          seq?[] =
          (new ret:^[ serverC__Service ]
          (server.createService![pidStr (+$ > " call(" pidStr ","
               (getType) "," (getOID) ", createService, [" "[]]") ret] |
                ret?[ sv ] =
          (new ret:^[ clientC__Client ]
          (c.createClient![pidStr (+$ > " call(" pidStr "," (getType)
                    "," (getOID) ", createClient, [" "[]]") ret] |
                ret?[ cl ] =
          (new ret:^[ ]
          (cl.use![pidStr (+$ > " call(" pidStr "," (getType) ","
                       (getOID) ", use, ["
                       "(in,sv,serverC__Service," (sv.getOID)
                       "), " "[]]") sv ret] |
                ret?[ ] =
          ()
          ))       ))       ))
          ))
          run initialized![(procId) ""]

     {- translation of THIS Structure -}
               [getOID = getOID
               getType = getType
               ]
))
```

Abbildung 12.3: Pict-Übersetzung von `csFramework`

12.1.1.2 Pict-Compiler

Pict ist ebenfalls nicht direkt ausführbar, sondern muss durch einen Compiler in ausführbaren Code (Executable) der jeweiligen Laufzeitumgebung übersetzt werden. In dieser Arbeit wurde wie erwähnt die Entwicklungsumgebung Pict 4.0 eingesetzt, die einen solchen Compiler zur Verfügung stellt, der als Komponente in den Simulator integriert wurde.

12.1.1.3 Laufzeitsystem

Das Laufzeitsystem ist im Wesentlichen das Ergebnis der Übersetzung des Pict-Compilers (siehe Abschnitt 12.1.1.2). Es ist ein direkt unter Sun-Solaris (Laufzeitumgebung) ausführbares Programm, das über die Standard-Unix-Pipes (`stdin` und `stdout`, siehe [Robbins 1999]) mit den anderen Komponenten kommuniziert.

12.1.1.4 Geschäftsprozessverwaltung

In der Geschäftsprozessverwaltung werden die im Rahmen der Konfiguration eines Anwendungssoftwareproduktes abgeleiteten Geschäftsprozesse protokolliert und verwaltet. Zusammen mit den Interaktionen der Geschäftsprozesse werden ebenfalls die jeweils angefallenen Konfigurationsentscheidungen protokolliert. Der aktuell zu bearbeitende Geschäftsprozess wird dem Benutzer zur Verfolgung des Geschäftsprozesses angezeigt. Ferner kann der Benutzer einzelne Geschäftsprozesse zum Löschen auswählen, wodurch dem Konfigurator die (inversen) Konfigurationsentscheidungen, die mit diesem Geschäftsprozess verbunden sind, zur Rücknahme mitgeteilt werden.

12.1.2 Konfigurator

Der Konfigurator (siehe Abbildung 12.4) verwaltet die aktuelle Konfiguration, berechnet zu einem Slot die Konfigurationsvarianten und bietet sie dem Simulator zur Auswahl an. Anderseits bekommt der Konfigurator Entscheidungen für ausgewählte Komponenten mitgeteilt, propagiert diese im System und berechnet dadurch die Konfiguration neu.

Als Konfigurationssystem wurde NEOCLASSIC von den Bell Labs (siehe [Borgida et al. 1989]) gewählt. NEOCLASSIC erfordert ein Lisp-ähnliches Eingabeformat, in das die CCL-Beschreibung eines Anwendungssoftwareprodukts vor der Konfiguration übersetzt werden muss. Dazu wurde im Rahmen dieser Arbeit der Übersetzter CCL2NeoClassic in der Programmiersprache PROLOG implementiert. Das durch den Compiler erzeugte NEOCLASSIC-Programm wird dann in das NEOCLASSIC-Laufzeitsystem – dies ist Bestandteil der Konfigurationsmaschine – geladen, die die Datei direkt interpretiert. Als Beispiel für die NEOCLASSIC-Definition dient wieder die Framework-Komponente `csFramework` aus Abschnitt 7.1. In Abbildung 12.5 ist die CCL2NeoClassic-Übersetzung der Komponentendefinition aus Abschnitt 9.3.3 zu sehen.

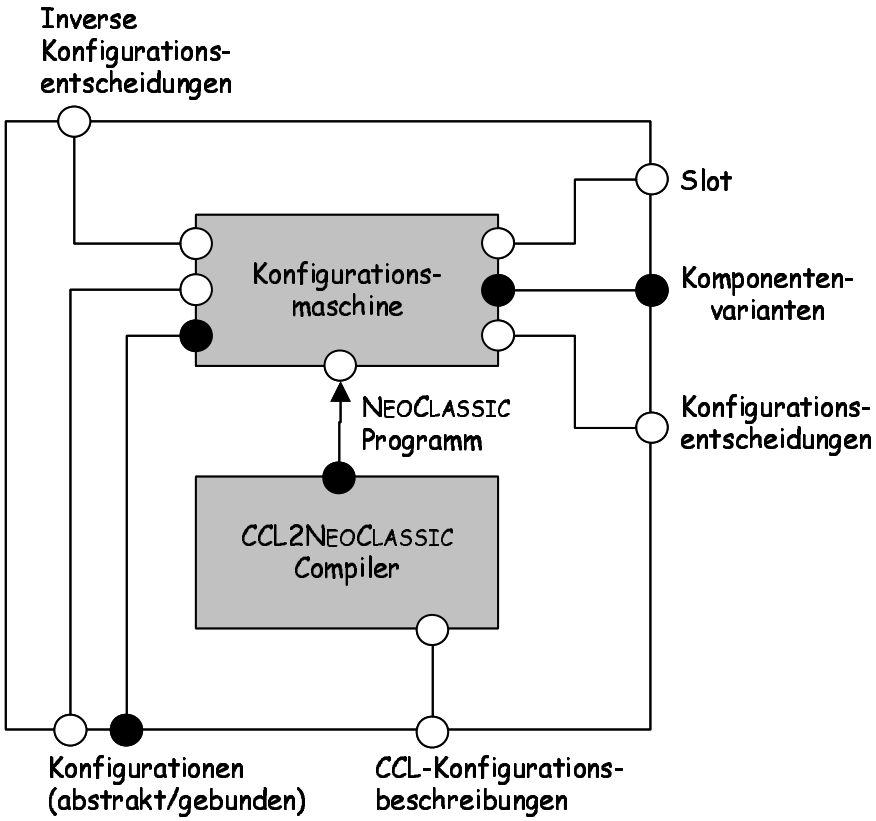


Abbildung 12.4: Die Konfiguratorkomponente

Zur Kommunikation mit den anderen Komponenten müssen im Wesentlichen zwei Befehle realisiert werden. Der erste Befehl ermittelt zu einem Slot die Menge der im Rahmen der Konfiguration erlaubten Komponenten. Der zweite Befehl dient der Eingabe einer Konfigurationsentscheidung. Die dazu notwendigen Funktionen werden im Folgenden erläutert.

```
;----------------- csFramework      -----------------
(createRole s true)
(createRole c true)

(createConcept csFramework
     (and
          COMPONENT
          (all s serverC)
          (atLeast 1 s)
          (all c clientC)
          (atLeast 1 c)
     )
     component-disjunction
)
```

Abbildung 12.5: NEOCLASSIC-Übersetzung von `csFramework`

12.1.2.1 Komponentenvarianten zu einem Slot

Ein Slot ist aus Sicht der Konfiguration ein Verweis auf eine abstrakte Komponente, die durch die Übersetzung zur Laufzeit als Individuum eines NEOCLASSIC-Konzeptes repräsentiert wird. Im Rahmen der Konfiguration wird diese Komponente zunehmend konkretisiert, d.h. die Menge der für eine konkrete Konfiguration in Frage kommenden Komponenten wird eingeschränkt. In NEOCLASSIC steht zur Ermittlung aller Konzepte, die ein Individuum umfassen (subsumieren) der Befehl `getParents` (siehe [Patel-Schneider et al. 1996]) zur Verfügung. Alle Konzepte, die ein Individuum subsumieren, repräsentieren dann alternative Softwarekomponenten, die in einen Slot eingesetzt werden können.

12.1.2.2 Eingabe von Konfigurationsentscheidungen

Konfigurationsentscheidungen werden durch die drei Konfigurationsoperatoren `addComponent`, `spezializeComponent` und `addReferenz` in das Konfigurationssystem eingegeben. Wie diese Operatoren in NEOCLASSIC implementiert werden, lässt sich unmittelbar aus den NEOCLASSIC-Beispielen ableiten, die den Definitionen (siehe Abschnitt 9.3.6) zur Veranschaulichung nachgestellt wurden. Die Rücknahme von Konfigurationsentscheidungen (inverse Konfigurationsentscheidungen) werden analog durch die Übersetzungen der Befehle `delComponent`, `generalizeComponent` und `delReferenz` realisiert.

12.1.3 Fachübersetzer

Der Fachübersetzer basiert auf der Idee der Substitution von Variablen durch SCDL-Instanzen – einschließlich ihrer Typbezeichner (Komponente oder Klasse). Die Substitution wurde in Abschnitt 8.1 definiert. Das sie logikbasiert ist, lässt sie sich direkt in PROLOG implementieren.

12.2 Persistenzebene

Als „Datenbanksystem“ dient bislang nur das Dateisystem des zugrunde liegenden Betriebssystems (Sun Solaris). Der Einsatz eines datenbankgestützten Repositories ist im Rahmen eines weiteren Projekts KOSOBAR (siehe [Sandmann und Ritter 2000]) geplant.

12.3 Benutzungsebene

Genauso wie die Persistenzebene wurde auch die Benutzungsebene nur rudimentär realisiert. Im Rahmen von PROSECCO sollte ursprünglich eine graphische Benutzungsoberfläche implementiert werden, die die interpretierten Interaktionen und die daraus abgeleiteten Geschäftsprozesse in Form von Diagrammen abbildet. Wie in Abbildung 10.3 veranschaulicht wurde, sollte hierzu eine EPK-ähnliche Notation genutzt werden. Der Zugang für den

Benutzer ist bislang jedoch nur über die verschiedenen Schnittstellen der eingesetzten Werkzeuge (Pict, NEOCLASSIC, PROLOG) möglich.

Ein entsprechendes Werkzeug, mit dem eine graphenbasierte Benutzungsoberfläche realisiert werden kann, wurde zwar mit ViaGra (siehe [Ritter und Dirichs 1998]) realisiert, es konnte jedoch aus Zeitgründen bislang nicht angebunden werden. Allerdings hat es sich in der Arbeit von Jaekel (siehe [Jaekel 1999]) für eine ähnliche Aufgabenstellung als praxistauglich erwiesen. ViaGra ist in der Programmiersprache Java (siehe [Flanagan 1997]) implementiert worden.

Teil IV Epilog

13 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, Methoden und Werkzeuge zu entwickeln, die den Anwender bei der modellgestützten Entwicklung komponentenbasierter betrieblicher Anwendungssysteme unterstützen sollen.

Dazu wurden im ersten Teil zuerst Hypothesen und Bedingungen aufgeführt, die den Rahmen für diese Arbeit bilden. Danach wurde ein Überblick über heutige betriebliche Anwendungssysteme und die Technologie zukünftiger komponentenbasierter Anwendungssysteme gegeben. Da als Grundlage zur Modellierung komponentenbasierter Anwendungssysteme kein ausschließlich formaler Ansatz verfolgt werden sollte, wurden zusätzlich die Sprachansätze (Norm- bzw. Orthosprachen) der konstruktiven Wissenschaftstheorie vorgestellt, die einen wesentlichen Eckpfeiler dieser Arbeit darstellen. Sie erlauben es, einem formalen System, z.B. einem betrieblichen Anwendungssystem, eine „inhaltliche“ Semantik zuzuordnen, die sich an der Semantik von natürlichen Sprachen anlehnt, und damit dem Anwender, der mit der Komplexität formaler Sprachen überfordert wäre, eine verständliche Grundlage zu bieten.

Im zweiten Teil, der den Kern der Arbeit darstellt, wurden Sprachen zur Beschreibung von Softwarekomponenten (SCDL) und zur Kombination dieser Komponenten zu größeren Anwendungssoftwareprodukten (CCL) entwickelt. Da die auf diesen Sprachen basierenden Beschreibungen grundsätzlich formalen Charakter haben, wurden zusätzlich Abbildungen zwischen solchen Beschreibungen und normsprachlichen Sätzen definiert, die es dem Anwender ermöglichen, mit quasinatürlichsprachlichen Begriffen und Aussageformen seine Interpretation eines Anwendungssystems in die Beschreibungen eines Anwendungssoftwareproduktes einzubringen. Durch diese Art der Interpretation werden dann formale Traces, die das beispielhafte Zusammenspiel mehrerer SCDL-Komponenten aufzeigen, zu Beschreibungen, die die mit der Software zukünftig realisierbaren Geschäftsprozesse darstellen. Zum Ende dieses Teils wird eine Methode vorgestellt, die die Konfiguration von Anwendungssystemen und dabei gleichzeitige Ableitung von Geschäftsprozessen (die sich als Konsequenz aus einer Konfiguration ergeben) erlaubt.

Im dritten Teil wird die prototypische Implementierung (PROSECCO) des erarbeiteten Ansatzes vorgestellt. Der vierte und fünfte Teil enthalten Übersichten sowie weitere Grundlagen zu dieser Arbeit.

Kritik

Das Thema der modellgestützten Konfiguration aus Sicht des Anwenders wird selbst bei den heute existierenden monolithischen betrieblichen Anwendungssystemen noch stark vernachlässigt. Um so schwerer ist es, klare Aussagen über Chancen und Risiken des hier

vorgestellten Ansatzes zu machen, zumal auch die Entwicklung komponentenbasierter Standards –sowohl technischer als auch inhaltlicher Art – erst am Anfang steht. Aus diesem Grund sind – und das sollte durch die aufgeführten Hypothesen im ersten Kapitel dieser Arbeit verdeutlicht werden – auch die Ergebnisse dieser Arbeit stark von der weiteren Entwicklung der Komponententechnologie abhängig. Allein die Frage, ob es in Zukunft überhaupt einen bedeutenden Komponentenmarkt geben wird, ist heute noch vollkommen offen.

Dennoch lassen sich – unabhängig von den geschilderten Rahmenbedingungen – einige Fragen und Kritikpunkte aufführen, die Ansatzpunkte für weitere Arbeiten bieten.

Suchen und Finden von Komponenten

Ein Grundproblem bei der Gestaltung komponentenbasierter Anwendungssysteme, das in dieser Arbeit vollkommen ignoriert wurde, ist das des Suchens und Findens (siehe [Biggerstaff und Richter 1987]) der „richtigen“ Komponente für ein Anwendungssoftware-produkt.

Heute bekannte Ansätze setzen meist auf textuellen Beschreibungen der Komponenten auf, die zu einer Komponente angegeben werden müssen und dann von Seiten der suchenden Akteure (Komponentenverwender) als Grundlage für Freitextrecherchen genutzt werden können. Jedoch können auch weitergehende Ansätze verfolgt werden, die formale Schnittstellensignaturen oder sogar Verhaltensbeschreibungen automatisch auf Ähnlichkeiten (z.B. formale Substituierbarkeit) hin untersuchen.

Substituierbarkeit von Komponenten

Die Frage, wie zu entscheiden ist, ob eine Komponente durch eine andere Komponente substituierbar ist, bzw. ob eine Komponente den Kontrakt eines Frameworks erfüllt, wurde in dieser Arbeit nur angerissen. Sie stellt jedoch einen wesentlichen Ausgangspunkt für „korrekte“ Anwendungssysteme dar, für die z.B. der Anwendungsarchitekt verantwortlich sein kann. Zwar wurde in dieser Arbeit auf ein formales Fundament für die konzipierten Beschreibungssprachen geachtet, dennoch wurde auf entsprechend formale Beweisverfahren nicht weiter eingegangen.

Tatsache ist, dass Ansätze zum automatischen Nachweis der Substituierbarkeit bei komplexen Systemen, wie sie betriebliche Anwendungssysteme nun einmal darstellen, heute kaum praktikabel einsetzbar sind. Werden jedoch Abstriche bei der Genauigkeit der Beschreibung von Komponenten hingenommen, wird also vom konkreten Verhalten „etwas“ abstrahiert, so gibt es durchaus Verfahren, die den Anwender bei der Suche und Überprüfung potentieller Komponente unterstützen können. Hierzu sei z.B. auf entsprechende Arbeiten (siehe z.B.

[Canal, Pimentel und Troya 1998], [Sangiorgi 1993] und [Pierce und Sangiorgi 1999]) verwiesen, die auf dem π-Kalkül basierende Verfahren für Substituierbarkeitstest entwickeln.

Verständlichkeit von Normsprache

Es wurde davon ausgegangen, dass Normsprachen für den Anwender „relativ leicht“ zu verstehen sind, da sie sich an der Syntax und Semantik germanischer Sprachen orientieren. Tatsache ist jedoch, dass diese Sprachen auch erst wieder gelernt werden müssen. Andererseits können Normsprachen als Ausgangspunkt für „richtige“ natürlichsprachliche Sätze genutzt werden (siehe [Lorenzen 1987]). Dazu müssen die Flexionen (Konjugationen und Deklinationen) der im Rahmen der Normsprache genutzten Prädikatoren zusätzlich in einem Lexikon verwaltet werden und bei der Präsentation normsprachlicher Sätze berücksichtigt werden. So können z.B. bei Anwendung eines deutschen Vokabulars automatisch deutschsprachige Sätze erzeugt werden. Diese Idee wird schon heute im Bereich des Maschinenbaus verfolgt, wo Handbücher und technische Manuale auf Basis von „kontrolliertem Englisch“ (controlled english) formuliert werden.

Implementierung

Das im Rahmen dieser Arbeit entwickelte Konfigurationssystem dient nur der exemplarischen Umsetzung der in der Arbeit entwickelten Konzepte. Es eignet sich jedoch nicht zur Konfiguration eines Anwendungssoftwareproduktes bei einem „richtigen“ Anwender. Zur Evaluation der vorgestellten Konzepte wäre es deshalb unumgänglich, eine vollständigere und softwareergonomisch bessere Implementierung zu entwickeln.

Fazit

Diese Arbeit stellt einen sehr breit angelegten Versuch dar, die heute relevanten Aspekte der geschäftsprozessorientierten und anwendungsbezogenen Konfiguration zukünftiger komponentenbasierter Anwendungssysteme unter einem Dach zu vereinigen. Viele der damit verbundenen Fragestellungen konnten deswegen nur angerissen werden, obwohl sie eine grundlegendere und weitergehende Betrachtung verdienen.

Grundsätzlich wird der hier vorgeschlagene Weg der geschäftsprozessorientierten Konfiguration komponentenbasierter Anwendungssysteme jedoch verteidigt. Denn es ist abzusehen, dass die Potentiale betrieblicher Anwendungssysteme aufgrund der Komponentenorientierung immer weiter zunehmen werden, es aber andererseits für den Anwender immer schwieriger wird, die Übersicht über die Gestaltungsmöglichkeiten seiner betrieblichen Anwendungssysteme zu behalten.

14 Abbildungsverzeichnis

Abbildung 1.1: Makroprozess komponentenbasierter Anwendungsentwicklung 6
Abbildung 1.2: Framework-basiertes Anwendungssystem 7
Abbildung 1.3: Prozessorientierte Konfiguration 14
Abbildung 1.4: Die Kapitel der Arbeit und ihre Abhängigkeiten 17
Abbildung 2.1: Teilsysteme des Unternehmens (siehe [Ferstl und Sinz 1998]) 19
Abbildung 2.2: Unternehmensarchitektur (siehe [Ferstl und Sinz 1998]) 20
Abbildung 2.3: Konfigurationsbaum 28
Abbildung 3.1: Framework 40
Abbildung 3.2: Komponenten vs. Komponenteninstanzen 47
Abbildung 3.3: Vermittlung zwischen sendenden und empfangenden Komponenten 52
Abbildung 3.4: Abhängigkeiten zwischen Komponenten (1) 52
Abbildung 3.5: Abhängigkeiten zwischen Komponenten (2) 53
Abbildung 3.6: UND/ODER-Graph (1) 56
Abbildung 3.7: UND/ODER-Graph (2) 57
Abbildung 4.1: Substituierbarkeit 64
Abbildung 4.2: Skizze der ARIS-Architektur 72
Abbildung 4.3: Beschreibungselemente des EPK-Modells 73
Abbildung 4.4: EPK-Beispiel 75
Abbildung 4.5: Zwei einfache Zustände mit einer Transition 80
Abbildung 4.6: Ein komplexer Zustand 80
Abbildung 4.7: Einfache Darwin-Komponente 85
Abbildung 4.8: Komplexe Darwin-Komponente 86
Abbildung 5.1: Gegenstandseinteilung 95
Abbildung 5.2: Wortarten 97
Abbildung 5.3: Satzarten 98
Abbildung 5.4: Beispiel eines Ableitungsbaumes (Tiefenstruktur) 103
Abbildung 6.1: Zusammenhang der Beschreibungen 117
Abbildung 7.1: Komponentenmodell 119
Abbildung 7.2: Beispiel für eine SCDL-Beschreibung 121
Abbildung 7.3: Schichtenarchitektur zum Beispiel 123
Abbildung 7.4: Trace zum Beispiel 142
Abbildung 8.1: Abstraktionsgrade im Gestaltungsprozess 143
Abbildung 8.2: Trace mit Zuordnung von Annotationen 154
Abbildung 9.1: Beispiel-Systemmodell 163
Abbildung 9.2: Subordinationsbeziehungen 165
Abbildung 9.3: Parametrierungsbeziehungen 166
Abbildung 9.4: Kompositionsbeziehungen 167
Abbildung 9.5: Graph des Konfigurationsraums (Beispiel) 174
Abbildung 10.1: Soll-Geschäftsprozesse 181
Abbildung 10.2: Wechselwirkungen mit Geschäftsprozessen 182
Abbildung 10.3: Trace in EPK-Darstellung 188
Abbildung 10.4: Konzepte der Konfiguration 190
Abbildung 10.5: Prozessorientierte Konfiguration (Grobsicht) 190
Abbildung 10.6: Algorithmus zur Konstruktion von Geschäftsprozessen 191
Abbildung 11.1: Grobarchitektur 198
Abbildung 11.2: Anwendungsebene 199

ABBILDUNG 12.1: SIMULATOR-KOMPONENTE 204
ABBILDUNG 12.2: PROLOG-REGEL DES SCDL2PICT-ÜBERSETZERS 204
ABBILDUNG 12.3: PICT-ÜBERSETZUNG VON csFramework 206
ABBILDUNG 12.4: DIE KONFIGURATORKOMPONENTE 208
ABBILDUNG 12.5: NEOCLASSIC-ÜBERSETZUNG VON csFramework 208

15 Tabellenverzeichnis

TABELLE 2.1: AUFGABEN UND TYPEN VON BETRIEBLICHEN ANWENDUNGSSYSTEMEN (BSP.) 24
TABELLE 4.1: BEWERTUNG VON SPRACHEN ZUR VERHALTENSBESCHREIBUNG VON KOMPONENTEN 87
TABELLE 5.1: SATZARTEN UND KOPULAE 99
TABELLE 6.1: ANFORDERUNGEN AN BESCHREIBUNGEN 117
TABELLE 18.1: SYNTAX FÜR KONZEPTBESCHREIBUNGEN 248
TABELLE 18.2: TERMINOLOGISCHE AXIOME 249
TABELLE 18.3: ASSERTIONALE AXIOME 249

16 Zeichenerklärungen

16.1 Abkürzungen

4GL	Fourth Generation Language
ABAP/4	Advanced Business Application Programming/4th Generation
ABox	assertionale Box
ACID	Atomicity, Consistency, Isolation, Durability
ADL	Architecture Description Language
$\mathcal{ALC}$	(Beschreibungslogik)
ARIS	Architektur integrierter Informationssysteme
BAPI	Business Application Programming Interface
BPR	Business Process Reengineering
CASE	Computer Aided Software Engineering
CCL	Component Configuration Language
CCL2DL	CCL-nach-DL-Übersetzer
CCS	Calculus of Communicating Systems
COM	Common Object Model
CORBA	Common Object Request Broker Architecture
CSP	Communicating sequential processes
DCOM	Distributed Common Object Model
DL	Description Logic
EBNF	Extended Backus Naur Form
ECA	Event, Condition, Action
EDI	Electronic Data Interchange
EDIFACT	EDI for Administration, Commerce and Transport
EJB	Enterprise JavaBeans
EPK	Ereignisgesteuerte Prozesskette
ERP	Enterprise Resource Planing
FIBU	Finanzbuchhaltung
GUI	Graphical User Interface
HJA	Hans-Jürgen Appelrath
IDES	International Demonstration and Education System
ISO	International Standards Organization
ISV	Independent Software Vendors
LISP	List processing Language
MAWI	Materialwirtschaft
OAG	Open Application Group
OAGIS	Open Applications Group Integration Specification
OMG	Object Management Group
PPS	Produktionsplanungs- und Steuerungssystem
PROSECCO	Process descriptions for the software engineer - consultant - customer - supply chain
RCS	Revision Control System
RHB	Roh-, Hilfs- und Betriebsstoffe
SCDL	Simple Component Description Language
SCM	Software Configuration Management
SQL	Structured Query Language

STEP	International Standard for the Computer-Interpretable Representation and Exchange of Product Data
TBox	terminologische Box
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language

16.2 EBNF

Zur Definition der formalen Syntax verschiedener Sprachen dieser Arbeit (SCDL, CCL, π-Kalkül) wird die Notation der erweiterten Backus-Naur-Form (EBNF) eingesetzt, mit deren Hilfe kontextfreie Grammatiken (siehe z.B. [Mayer 1986]) definiert werden können. Eine kontextfreie Grammatik besteht aus einer Menge von Nichtterminalen (Ableitungssymbolen), einer Menge von Terminalen (Worten der Sprache), einer Menge von Produktionen und einem Startsymbol aus der Menge der Nichtterminale. Produktionen (Ableitungsregeln) bestehen aus einem Kopf und einem Rumpf. Der Kopf ist ein Nichtterminal und bezeichnet die Produktion, und der Rumpf definiert mögliche Ableitungen/Ersetzungen dieses Nichtterminals. Die erste Ableitung muss ausgehend vom Startsymbol vorgenommen werden. In dieser Arbeit wird das Startsymbol dadurch gekennzeichnet, dass es das erste Symbol (Kopf) der ersten Produktion einer Grammatik ist. Die Konstruktion eines Satzes bzw. Wortes aus dieser Sprache beginnt also mit der ersten Regel einer Grammatik.

Eine (kontextfreie) Grammatik G definiert die Syntax einer Sprache. Die Menge L(G) aller syntaktisch richtigen – aber nicht notwendigerweise semantisch sinnvollen – Sätze einer solchen Sprache kann durch die Grammatik durch systematisches Anwenden der Produktionen ausgehend vom Startsymbol (bzw. von der ersten Produktion) erzeugt werden.

Ausgehend von allen Nichtterminalen der Grammatik G können Mengen von (Teil)-Sätzen abgeleitet werden. Die Menge der Teilsätze eines Nichtterminals N wird dann auch oft mit L(N) bezeichnet. In einigen Abschnitten dieser Arbeit wird von dieser Schreibweise Gebrauch gemacht.

Die verwendeten Symbole der EBNF-Notation haben folgende Bedeutung:

`::=`	Definition: Dieses Symbol definiert eine Produktion und trennt dabei Produktionskopf und –rumpf.
`<...>`	Nichtterminale: Nichtterminale sind Symbole, die durch spitze Klammern eingeschlossen sind: Beispiel: <a>, <procedure>, <loop>.
`|`	Alternativen: Alternativen innerhalb einer Produktion werden durch dieses Zeichen voneinander getrennt.
`(...)`	Klammerung: Die Klammerung dient der Eindeutigkeit von Ableitungen.

[...]	Optionalität: In diese eckigen Klammern eingeschlossene Symbole (Nichtterminale und Terminale) können optional zu einer Ableitung hinzu genommen werden.
{...}	Beliebige Wiederholung: In die geschweiften Klammern werden Symbole eingeschlossen, die keinmal, einmal oder beliebig oft Teil der Ableitung sein dürfen.
$\{...\}^+$	Ein- und mehrfache Wiederholung: In die geschweiften Klammern mit angehängten Pluszeichen werden Symbole eingeschlossen, die einmal oder beliebig oft Teil einer Ableitung sein dürfen.
.	Ende der Produktion: Das Ende einer Produktion wird durch einen Punkt gekennzeichnet.

Alle Terminale einer Sprache, die den Zeichen der EBNF-Notation (beispielsweise „|“, „[“, „]“) entsprechen, müssen, um Verwechslungen vorzubeugen, zusätzlich durch Anführungsstriche („“) gekennzeichnet werden.

Literaturverzeichnis

Abinava, S.; Buch, A.; Cattior, E.; Chilant, M.; El-Rafei, S.; Krause, A.; Müller, W. und Zulian, F. 1998. *San Francisco Concepts & Facilities (DOCNUM SG24-2157-00)*. Hrsg.: IBM, *Redbook*. Rochester: IBM.

Agha, G. und Hewitt, C. 1987. Actors: a conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, Hrsg.: B. Shriver und P. Wegner. Cambridge: MIT Press.

Appelrath, H.-J. und Ritter, J. 1999. *R/3-Einführung - Methoden und Werkzeuge*. Berlin u.a.: Springer-Verlag.

Baader, F.; Bürckert, H.-J.; Hollunder, B.; Laux, A. und Nutt, W. 1992. Terminologische Logiken. *Künstliche Intelligenz* 6 (3):23-33.

Baader, F. und Hollunder, B. 1991. KRIS: Knowledge Representation and Inference System. *ACM SIGART Bulletin* 2 (3):8-14.

Bach, C. 1994. *An Interactive Knowledge-Based Shell for Configuration Tasks*. Band 34, *Series in Micro-electronics*. Konstanz: Hartung-Gorre Verlag.

Back, R. J. R. und Wright., J. v. 1998. *Refinement Calculus: A Systematic Introduction*. Berlin u.a.: Springer-Verlag.

Berlack, H. R. 1992. *Software Configuration Management*. New York: John Wiley & Sons.

Bibel, W. 1993. *Wissenrepräsentation und Inferenz: eine grundlegende Einführung*. Braunschweig: Vieweg.

Biggerstaff, T. J. und Richter, C. 1987. Reusability framework, assessment, and directions. In *Software Reusability*, Hrsg.: T. J. Biggerstaff und A. Perlis. New York: ACM Press.

Booch, G. 1994. *Object-Oriented Analysis and Design*. Hrsg.: G. Booch. 2. Auflage, *The Benjamin/Cummings Series in Object Oriented Software Engineering*. Santa Clara: The Benjamin/Cummings Publishing Company, Inc.

Booch, G.; Rumbaugh, J. und Jacobson, I. 1997a. Unified Modeling Language - Notation Guide Version 1.1 . RATIONAL Software Corporation, Santa Clara.

Booch, G.; Rumbaugh, J. und Jacobson, I. 1997b. Unified Modeling Language - Semantics Version 1.1 . RATIONAL Software Corporation, Santa Clara.

Borgida, A.; Brachman, R. J.; McGuinness, D. L. und Resnick, L. A. 1989. Classic: A Structural Data Model for Objects. ACM SIGMOD International Conference on Management of Data, Portland.

Borgida, A. und Patel-Schneider, P. F. 1994. A Semantics and Complete Algorithm for Subsumption in the CLASSIC Description Logic. *Journal of Artificial Intelligence Research* 94 (1):277-308.

Brachmann, R. J. und Schmolze, J. G. 1985. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science* 9 (2):171-216.

Breu, R.; Hinkel, U.; Hofmann, C.; Klein, C.; Paech, B.; Rumpe, R. und Thurner, V. 1997. Towards a Formalization of the Unified Modeling Language. ECOOP '97.

Canal, C.; Pimentel, E. und Troya, J. M. 1998. Compatibility, inheritance and extension of PI-calculus agents. Málaga: Universidad de Málaga.

Chen, R. und Scheer, A.-W. 1994. Modellierung von Prozessketten mittels Petri-Netz-Theorie. Saarbrücken: Institut für Wirtschaftsinformatik an der Universität des Saarlandes.

Church, A. 1941. *The Calculi of Lambda Conversion*: Princeton University Press.

Clocksin, W. F. und Mellish, C. S. 1987. *Programming in PROLOG*. Berlin u.a.: Springer-Verlag.

Dittrich, J.; Mertens, P. und Hau, M. 1999. *Dispositionsparameter von SAP R/3-PP*. Braunschweig u.a.: Vieweg.

Ebert, J. und Engels, G. 1995. Specialization of Object Life Cycle Definitions. Koblenz: Fachbericht Informatik.

Engesser, H.; Claus, V. und Schwill, A. 1993. *Duden Informatik*. Mannheim: Dudenverlag.

Feiler, P. H. 1991. Configuration Management Models in Commercial Environments. Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Ferstl, O. K. und Sinz, E. J. 1998. *Grundlagen der Wirtschaftsinformatik*. Band 1 von 2. Wien: Oldenbourg Verlag.

Flanagan, D. 1997. *Java in a Nutshell*. 2. Auflage. Cambridge u.a.: O'Reilly.

Fowler, M. 1997. *Analysis patterns: reusable object models*. Menlo Park, Californien: Addison-Wesley.

Fowler, M. und Scott, K. 1998. *UML konzentriert*. Bonn u.a.: Addison Wesley Longman.

Friedman, D. P. und Felleisen, M. 1989. *The little Lisper*. 3. Auflage. Chicago u.a.: Science Research Assoc.

Gamma, E.; Helm, R.; Johnson, R. und Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley.

Griffel, F. 1998. *Componentware - Konzepte und Techniken eines Softwareparadigmas*. Heidelberg: dpunkt-Verlag.

Habermann, H.-J. und Leymann, F. 1993. *Repository: eine Einführung*. München u.a.: Oldenbourg.

Hammer, M. und Champy, J. 1994. *Business reengineering: die Radikalkur für das Unternehmen*. Frankfurt (Main) u.a.: Campus-Verlag.

Harel, D. und Gery, E. 1997. Executable Object Modeling with Statecharts. *Computer: a publication of the IEEE Computer Group* 30 (7).

Harel, D. und Naamad, A. 1996. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering Methodology*. 5 (4).

Hartmann, D. 1990. *Konstruktive Fragelogik*. Mannheim, Wien, Zürich: BI-Wissenschaftsverlag.

Hoare, C. A. R. 1985. *Communicating sequential processes*. New York u.a.: Prentice Hall.

Honda, K. und Tokoro, M. 1991. An object calculus for asynchronus communication. In *ECOOP 91*. Berlin u.a.

Hufgard, A. 1994. *Betriebswirtschaftliche Softwarebibliotheken und Adaption*. München: Vahlen.

Hümpel, C.; Renner, T. und Schmitz, V. 1999. *Spezifikation BMEcat* (1.01) BME, 1999 [Zugriff am 10.01.1999]. URL: http://www.bme.de.

IBIS Prof. Thome GmbH. 1999. *LIVE KIT Structure* IBIS Prof. Thome GmbH, 1999 [Zugriff am 28.10.1999]. URL: http://www.ibis-thome.de/products/livekit/index.html.

Jablonski, S. und Bussler, C. 1996. *Workflow Management*. London: Thomson Press.

Jacobson, I. 1998. *Object-oriented software engineering: a use case driven approach*. Harlow u.a.: Addison-Wesley.

Jaekel, H. 1999. Konfiguration von framework-basierten Anwendungen. Diplomarbeit, Fachbereich Informatik, Abteilung Informationssysteme, Carl v. Ossietzky Universität Oldenburg, Oldenburg.

Jaeschke, P. 1995. Geschäftsprozessmodellierung mit INCOME. In *Geschäftsprozeßmodellierung und Workflow-Management*, Hrsg.: J. Becker und G. Vossen. Bonn: International Thompson Publishing.

Kappel, G.; Retschitzegger, W. und Schwinger, W. 1998. A Comparison of Role Mechanisms in Object-Oriented Modeling. Modellierung '98, Münster.

Keller, G. und Teufel, T. 1997. *SAP R/3 prozeßorientiert anwenden, Edition SAP*: Addison-Wesley.

Kilov, H. und Rumpe, B. 1997. ECOOP '97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques. München: Technische Universität München.

Kirchmer, M. 1998. *Business Process Oriented Implementation of Standard Software*. Berlin u.a.: Springer-Verlag.

Kutschera, F. v. 1976. *Einführung in die intensionale Semantik*. Berlin, New York: Walter de Gruyter.

Langer, G. 1994. Objektorientierte Unternehmensmodellierung mit AENEIS. Stuttgart: ipro Tool GmbH.

Langner, P.; Schneider, C. und Wehle, J. 1997. Ereignisgesteuerte Prozessketten und Petri-Netze. Hamburg: Universtität Hamburg.

Lechtenbörger, J. 1997. Geschäftsprozeßmodellierung unter Verwendung der Prozeßalgebra CCS. Diplomarbeit, Fachbereich Informatik, Universität Oldenburg, Oldenburg.

Levi, F. 1996. A Process Language for Statecharts. LOMAPS '96.

Loos, P. und Allweyer, T. 1998. Process Orientation and Object-Orientation - An Approach for Integrating UML and Event-Driven Process Chains (EPC). Saarbrücken: Institut für Wirtschaftsinformatik der Universität des Saarlandes.

Lorenzen, P. 1987. *Lehrbuch der konstruktiven Wissenschaftstheorie*. Mannheim, Wien, Zürich: BI-Wissenschaftsverlag.

Lumpe, M. 1999. A Pi-Calculus Based Approach for Software Composition, Institut für Informatik und angewandte Mathematik, Universität Bern, Bern.

Magee, J.; Dulay, N.; Eisenbach, S. und Kramer, J. 1995. Specifying Distributed Software Architectures. The 5th European Software Engineering Conference (ESEC '95), Sitges.

Matena, V. und Hapner, M. 1999. Enterprise JavaBeans Specification. Palo Alto: Sun Microsystems.

Mattsson, M. 1996. Object-Oriented Frameworks - A survey of methodological issues. Lund: Department of Computer Science.

Mayer, O. 1986. *Syntaxanalyse*. 3. Auflage. Mannheim u.a.: BI-Wissenschaftsverlag.

McGuinness, D. L. und Wright, J. R. 1998. An Industrial Strength Description Logic-based Configurator Platform. *IEEE Intelligent Systems* 13 (4):69-77.

Mertens, P.; Bodendorf, F.; König, W.; Picot, A. und Schumann, M. 1996. *Grundzüge der Wirtschaftsinformatik, Springer Lehrbuch*. Berlin: Springer-Verlag.

Meyer, B. 1988. *Object-Oriented Software Construction*. New York u.a.: Prentice Hall.

Milner, R. 1989. *Communication and concurrency*. New York u.a.: Prentice-Hall.

Milner, R.; Parrow, J. und Walker, D. 1992a. A calculus of mobile processes, I. *Information and Computation* 100 (1):1-40.

Milner, R.; Parrow, J. und Walker, D. 1992b. A calculus of mobile processes, II. *Information and Computation* 100 (1):41-77.

Mittelstraß, J. und Blasche, S. 1980-1996. *Enzyklopädie Philosophie und Wissenschaftstheorie*. Hrsg.: J. Mittelstraß. Mannheim, Wien, Zürich: BI-Wissenschaftsverlag.

Möhle, S. 1997. Kann man komplexe Anwendungen aus Microsoft-Komponenten gewinnen? In *Modellbasierte Einführung betrieblicher Anwendungssysteme*, Hrsg.: E. Klockhaus und H.-J. Scheruhn. Wiesbaden: Gabler-Verlag.

Neurath, O. 1979. *Wissenschaftliche Weltauffassung, Sozialismus und logischer Empirismus*. Frankfurt/ M.: Suhrkamp Verlag.

Nierstrasz, O.; Gibbs, S. und Tsichritzis, D. 1992. Component-oriented software development. *Communications of the ACM* 35 (9):160-165.

OAG. 1999. *Open Applications Group Integration Specifications (OAGIS)* (6.1) OAG, 1999 [Zugriff am 14.07.1999]. URL: http://www.openapplications.org.

Olderog, E.-R. 1991. *Nets, Terms and Formulars*. Cambridge u.a.: Cambridge University Press.

Olderog, E.-R. und Steffen, B. 1999. *Correct System Design*. Band LNCS 1710. Berlin u.a.: Springer-Verlag.

Ortner, E. 1997. *Methodenneutraler Fachentwurf*. Hrsg.: D. Ehrenberg, D. Seibt und W. Stucky, *Wirtschaftsinformatik*. Stuttgart, Leipzig: B. G. Teubner Verlag.

Patel-Schneider, P. F.; Abrahams, M.; Resnick, L. A.; McGuinness, D. L. und Borgida, A. 1996. *NeoClassic Reference Manual: Version 1.0* AT&T Labs Research, 1996 [Zugriff am 29.10.1999].
URL: http://www.research.att.com/sw/tools/classic/papers/NeoRef.html.

Pierce, B. und Sangiorgi, D. 1999. Behavioral Equivalence in the Polymorphic Pi-Calculus: University of Pennsylvania.

Pierce, B. C. und Turner, D. N. 1994. Concurrent Objects in a Process Calculus. Theory and Practice of Parallel Programming (TPPP), 1995, Sendai.

Pierce, B. C. und Turner, D. N. 1997. Pict: A Programming Language Based on the Pi-Calculus: Indiana University.

Pinson, L. J. und Wiener, R. S. 1991. *Objective-C: Object-Oriented Programming Techniques*. Reading: Addison-Wesley.

Pomberger, G. und Blaschek, G. 1993. *Software Engineering: Prototyping und objektorientierte Software-Entwicklung*. München: Carl Hanser Verlag.

Quine, W. V. O. 1960. *Word and Object*. Cambridge: M.I.T. Press.

Reisig, W. 1986. *Petri Netze - Eine Einführung*. Berlin u.a.: Springer-Verlag.

Remme, M. und Scheer, A.-W. 1996. Konstruktion von Prozeßmodellen. Saarbrücken: Institut für Wirtschaftsinformatik.

Resnick, L. A.; Patel-Schneider, P. F.; McGuinness, D. L.; Weixelbaum, E.; Abrahams, M. K.; Borgida, A.; Brachman, R.; Isbell, C. L. und Zalondek, K. C. 1996. *NeoClassic User's Guide: Version 0.7* AT&T Labs Research, 1996 [Zugriff am 29.10.1999]. URL: http://www.research.att.com/sw/tools/classic/papers/NeoGuide/guide.html.

Ritter, J. 1998a. Integration von Entwicklung und Einführung betrieblicher Standardanwendungssysteme durch Softwareproduktmodelle. Modellierung '98, Münster.

Ritter, J. 1998b. PROSECCO - Eine Methode zur modellbasierten Anwendungssystemgestaltung. Business Information Systems '98, Poznan.

Ritter, J. und Dirichs, M. 1998. Viagra - Visualisierung abstrakter Graphbeschreibungen. Oldenburg: OFFIS.

Robbins, A. 1999. *Unix in a Nutshell: A Desktop Quick Reference for System V Release 4 and Solaris 7, Nutshell Handbook*. Cambridge u.a.: O'Reilly.

Rumbaugh, J. 1991. *Object-oriented modeling and design*. Englewood Cliffs u.a.: Prentice-Hall International.

Rump, F. 1999. *Geschäftsprozeßmanagement auf der Basis ereignisgesteuerter Prozeßketten, Wirtschaftsinformatik*. Stuttgart u.a.: B.G. Teubner Verlag.

Rumpe, B. 1997. Formale Methodik des Entwurfs verteilter objektorientierter Systeme, Lehrstuhl Programmiermethodik, verteilte Systeme, Technische Universität München, München.

Sandmann, C. 1998. Normsprachliche Beschreibung des dynamischen Verhaltens von Objektsystemen. Diplomarbeit, Abteilung Informationssysteme des Fachbereichs Informatik, Carl von Ossietzky Universität Oldenburg, Oldenburg.

Sandmann, C. 1999. Ein Vorgehensmodell für die komponentenbasierte Softwareentwicklung auf Basis von Referenzmodellen. Oldenburg: OFFIS.

Sandmann, C.; Teschke, T. und Ritter, J. 2000. Ein Vorgehensmodell für die komponentenorientierte Anwendungsentwicklung. In *Information als Erfolgsfaktor - 2. Liechtensteinisches Wirtschaftsinformatik-Symposium an der FH Liechtenstein*. Hrsg.: B. Britzelmaier und S. Geberl. Stuttgart, Leipzig: B. G. Teubner Verlag.

Sangiorgi, D. 1993. A Theory of Bisimulation for the Pi-Calculus: University of Edinburgh.

Scheer, A.-W. 1998a. *ARIS - Modellierungsmethoden, Metamodelle, Anwendungen*. 3. Auflage. Berlin: Springer-Verlag.

Scheer, A.-W. 1998b. *ARIS - Vom Geschäftsprozeß zum Anwendungssystem*. 3. Auflage. Berlin: Springer-Verlag.

Schienmann, B. 1997. *Objektorientierter Fachentwurf*. Hrsg.: J. Buchmann et al. Band 20, *Teubner-Texte zur Informatik*. Stuttgart, Leipzig: B. G. Teubner Verlag.

Schmidt-Schauß, M. und Smolka, G. 1991. Attributive concept description with complements. *Artificial Intelligence* (48).

Schöning, U. 1987. *Logik für Informatiker*. Mannheim, Wien, Zürich: B.I. Wissenschaftsverlag.

SUN. 1997. *JavaBeans*: Sun Microsystems. Specification.

Szyperski, C. 1998. *Component Software*. Harlow u.a.: Addison-Wesley-Longman.

Tallman, O. und Kain, J. B. 1998. COM versus CORBA: A Decision Framework. *Distributed Computing* (September-December).

Tan, L. und Katayama, T. 1989. Meta Operations for Type Management in Object-Oriented Databases - A Lazy Mechanism for Schema Evolution. DOOD '89, Kyoto.

Tichy, W. F. 1985. RCS - A System for Version Control. *Software - Practice and Experience* 15 (7).

Van Hentenryck, P. 1989. *Constraint Satisfaction in Logic Programming*. Cambridge u.a.: The MIT Press.

Walker, D. 1995. Objects in the PI-calculus. *Information and Computation* 116 (2):253-271.

Weck, W. 1997. Independently Extensible Component Frameworks. In *Special Issues in Object-Oriented Programming - ECOOP '96 Workshop Reader*, Hrsg.: M. Mühlhäuser. Heidelberg: dpunkt Verlag.

Wedekind, H. 1992. *Objektorientierte Schema-Entwicklung*. Hrsg.: K. H. Böhling, U. Kulisch und H. Maurer. Band 85, *Reihe Informatik*. Mannheim u.a.: BI-Wissenschaftsverlag.

Weikum, G.; Wodtke, D.; Kotz-Dittrich, A.; Muth, P. und Weißenfels, J. 1997. Spezifikation, Verifikation und verteilte Ausführung von Workflows in MENTOR. *Informatik-Forschung und Entwicklung* 12 (2):61-71.

Wendt, S. 1998. *Programmzweck versus Programmform* [HTML-Seite]. Universität Kaiserslautern, 1998 [Zugriff am 15.06.1999]. URL: http://www-agwendt.e-technik.uni-kl.de/WWW-AG-Wendt/mosaiksteine/Zweck_Form.html.

Winter, A.; Winter, A.; Becker, K.; Bott, O. J.; Gräber, S.; Haas, P.; Hasselbring, W.; Haux, R.; Penger, O.-S.; Prokosch, H.-U.; Ritter, J.; Schütte, R. und Wünnemann, J. 1998. Referenzmodelle für die Unterstützung des Managements von Krankenhausinformationssystemen. GMDS '98, Bremen.

Yang, Z. und Duddy, K. 1996. CORBA: A Platform for Distributed Object Computing (A State-of-the-Art Report on OMG/CORBA). Queensland: University of Queensland.

Zeller, A. 1997. Configuration Management with Version Sets, Abteilung Softwaretechnologie, Technische Universität Braunschweig, Braunschweig.

Teil V Anhang

17 Der π-Kalkül

In diesem Kapitel werden die in dieser Arbeit benutzten π-Kalküle vorgestellt. Dabei handelt es sich zum einen um den ursprünglichen, in [Milner, Parrow und Walker 1992a] und [Milner, Parrow und Walker 1992b] vorgestellten, synchronen Kalkül, in dem sowohl Empfänger als auch Sender warten, bis eine Interaktion zwischen ihnen abgeschlossen wurde. Er liegt auch den meisten Arbeiten zum Thema „Architekturbeschreibungssprachen“ (siehe Abschnitt 4.4) zugrunde. Zum anderen wird die Sprache *Pict* (siehe [Pierce und Turner 1997]) vorgestellt, die auf eine asynchronen Variante des π-Kalküls zurückgeht. Pict dient als Grundlage für das Komponentenmodell dieser Arbeit.

17.1 Synchroner π-Kalkül

In diesem Abschnitt werden Syntax (Abschnitt 17.1.1) und Semantik (Abschnitt 17.1.2) des synchronen π-Kalküls beschrieben.

17.1.1 Syntax des synchronen π-Kalküls

Zunächst wird die Syntax des synchronen π-Kalküls in EBNF-Notation (siehe Abschnitt 16.2) angegeben. Der π-Kalkül ist ein elementarer Kalkül zur Beschreibung und Analyse paralleler Systeme mit einer dynamischen Kommunikationsstruktur. Ein System in diesem Kalkül ist eine Sammlung von unabhängigen Prozessen, die über Kommunikationskanäle miteinander kommunizieren. Kanäle werden durch einen Namen (`<ID>`) referenziert. Namen sind die einfachsten Bausteine dieser Sprache, sie haben keine Struktur. Es gibt eine beliebig große Menge von Namen, die hier mit kleinen Buchstaben dargestellt werden.

Sequentielle Prozesse (`<PROC>`) werden rekursiv auf Basis elementarer Aktionen (`<ACT>`) aufgebaut[1]. Es gibt zwei elementare Aktionen: eine zum Schreiben und eine zum Lesen. Der Operator „!“ kennzeichnet Schreiboperationen (siehe nachfolgende Grammatikproduktionen). Der Name `<ID1>` wird auf den Kanal `<ID2>` geschrieben, und danach wird mit dem Prozess `<PROC>` fortgefahren. Der Operator „?“ kennzeichnet Leseoperationen. Von dem Kanal `<ID1>` wird ein Name gelesen, an den Namen `<ID2>` gebunden, und dann wird mit dem Prozess `<PROC>` fortgefahren. In `<PROC>` werden zuvor alle Vorkommen von `<ID2>` durch den gelesenen Namen ersetzt.

---

[1] Es wird hier darauf verzichtet, die originären Symbole des π-Kalküls, wie sie bei [Milner, Parrow und Walker 1992a] genutzt werden, einzuführen. Statt dessen wird schon hier auf die Schreibweise abgezielt, die auch in der Sprache Pict genutzt wird.

```
<ACT>          ::=  <ID1>!<ID2>“.”<PROC>
                |   <ID1>?<ID2>“.”<PROC>.
```

Eine Menge alternativer Aktionen mit anhängenden Prozessen kann jeweils durch den Operator „+“ (bzw. Σ) voneinander getrennt werden. Wenn n = 0, dann wird dieser Prozess als leerer Prozess „()“ verstanden, wodurch auch das Ende eines sequentiellen Prozesses definiert ist. Die parallele Komposition zweier Prozesse wird durch das Zeichen „|“ symbolisiert. Durch den Restriktionsoperator („new“) wird ein neuer Name mit dem Gültigkeitsbereich (Scope) <PROC> eingeführt, der alle freien Vorkommen des neuen Namens in <PROC> bindet. Durch den Replikationsoperator[1] „!“ wird eine beliebige Anzahl von Kopien von <PROC> parallel komponiert. Er kann durch die rekursive Gleichung !PROC ≡ PROC | !PROC beschrieben werden. Der Operator dient der Darstellung von *Prozessagenten*, die mit Prozeduren imperativer Programmiersprachen vergleichbar sind. Ein Prozessagent kann dann als ein Kanal verstanden, auf dem immer wieder neue Argumente zur Verarbeitung geschickt werden können.

Alle aufgezählten Operatoren werden durch die folgenden Produktionen in ihrer Syntax definiert:

```
<PROC>         ::=  “(” <PROC> “)”
                |   Σ_{i∈{1..n}} <Act_i>
                |   <PROC> “|” <PROC>
                |   “(” new y “)” <PROC>
                |   !<PROC>.
```

17.1.2 Semantik des synchronen π-Kalküls

Die operationale Semantik des π-Kalküls wird typischerweise in Form von Reduktionsregeln angegeben. Die meisten der vorgestellten Operatoren erklären sich von selbst, so dass hier nur auf die besonders wichtige Kommunikationsregel eingegangen werden soll, die die Kommunikation zwischen zwei Prozessen beschreibt:

$$(... + x?y.P_1 + ...) \mid (... + x!z.P_2 + ...) \rightarrow P_1\{z/y\} \mid P_2$$

Wenn durch einen Prozess auf dem Kanal x der Name z geschickt wird und ein anderer Prozess auf diesem Kanal auf Informationen wartet, reduziert sich der Gesamtprozess auf den Prozess $P_1 \mid P_2$, wobei in P_1 alle freien Vorkommen von y durch z ersetzt werden. Diese Operation ist kommutativ und assoziativ. Die genaue und ausführliche Definition der formalen Semantik ist z.B. in [Milner, Parrow und Walker 1992a; Milner, Parrow und Walker

[1] Der Replikationsoperator unterscheidet sich syntaktisch vom Schreiboperator, da er ein einstelliger Prefixoperator und nicht ein zweistelliger Infixoperator ist.

1992b] nachzulesen. Es existieren mittlerweile einige Varianten des π-Kalküls. So existiert neben dem oben vorgestellten monadischen auch ein polyadischer Kalkül, in dem nicht nur einzelne Namen, sondern auch Tupel von Namen kommuniziert werden können.

17.2 Pict

In diesem Abschnitt wird zuerst die Kernsprache von Pict (*Core Pict*) vorgestellt, die eine asynchrone Variante des π-Kalküls darstellt und auf der die gesamte Sprache Pict aufbaut. Die Grundidee ist dabei, dass alle komplexeren Sprachkonzepte aus Pict wie z.B. abstrakte Datentypen oder funktionale Programmkonstrukte letztlich wieder auf die Kernsprache zurückgeführt werden können. Durch dieses Prinzip ist das formale Fundament von Pict und letztlich auch des Komponentenmodells dieser Arbeit (siehe Kapitel 9), das sich auf Pict stützt, begründet.

17.2.1 Unterschiede zum synchronen π-Kalkül

Ähnlich wie im synchronen π-Kalkül gibt es elementare Aktionen zum Lesen und Schreiben sowie weitere Operatoren zum Aufbau komplexer Prozesse. Wesentliche Unterschiede zwischen dem klassischen π-Kalkül und Core Pict liegen in folgenden Eigenschaften:

- Asynchronität: Core Pict ist asynchron, das heißt, ein Prozess, der Daten über einen Kanal gesendet hat, wartet nicht, bis der Empfänger diese Daten empfangen hat. Eine Empfangsbestätigung muss – wenn gewünscht – explizit durch ein zurückgeschicktes „Acknowledge“ realisiert werden. Dazu wird i.d.R. beim Senden ein *Return-Kanal* mitgegeben, auf dem eine Bestätigung erfolgen muss.
- Typisierung: Alle Kanäle sind typisiert und können somit nur Werte aufnehmen, die auch dem zu diesem Kanal angegeben Typ entsprechen.
- Primitive Datentypen: Es werden primitive Datentypen wie `String`, `Int` und `Bool` zur Verfügung gestellt, damit sie nicht explizit codiert werden müssen. In dieser Arbeit wird nur auf den Datentyp `Bool` zurückgegriffen.
- Komplexe Datentypen: Es können komplexe Datentypen aus einfacheren Datentypen aufgebaut werden. Neben Tupeln können auch Strukturen definiert werden. Strukturen sind Tupel mit Feldern, die durch einen Identifizierer (`Id`) referenziert werden können. Strukturen und Tupel können auch wieder andere Strukturen und/oder Tupel enthalten. Strukturelemente werden allgemein über Pfadausdrücke (`Path`), wie sie auch in Programmiersprachen wie Modula2 und Java üblich sind, angesprochen. Sowohl primitive als auch komplexe Datentypen können zur abkürzenden Schreibweise im Deklarationsteil (`Dec`) durch eine Typdefinition (`type`) eingeführt werden.

- Definition von Prozessagenten: Prozessagenten, wie sie durch den Replikationsoperator im synchronen Prozesskalkül realisiert werden, werden in Core Pict durch das Schlüsselwort `def` definiert.

Die komplexen Datentypen können genauso kommuniziert werden wie einfache Namen im ursprünglichen synchronen Kalkül. Dabei können die Namen auf Empfängerseite – sie sollen im Weiteren auch *Variablen* heißen – in Form eines Musters typisiert sein. Es werden nur dann gesendete Daten empfangen, wenn sie dem Muster auf Empfängerseite entsprechen („matchen“). In wohlgetypten Sprachen kann diese Übereinstimmung zum Übersetzungszeitpunkt statisch geprüft werden. Ein gesendetes Datum `v` *matcht* ein Muster `p`, die beide aus einer Reihe von (evtl. durch Label l_i gekennzeichnete) Elementen bestehen, wenn es eine gültige *Substitution* von Namen gibt, die wie folgt definiert ist:

$$\texttt{match(x:T, v)} = \{v \mapsto x\},\ \texttt{v ist vom Typ T.}$$
$$\texttt{match}([l_1\ p_1 \ldots l_n\ p_n], [l_1\ v_1 \ldots l_n\ v_n \ldots]) = \texttt{match}(p_1, v_1) \cup \ldots \cup \texttt{match}(p_n, v_n).$$

Wenn `v` und `p` nicht die gleiche Struktur haben, so ist *match* undefiniert.

Es sei darauf hingewiesen, dass durch die zweite Gleichung eine Art von Subtypbeziehung eingeführt wird, da konkrete Strukturen, die mehr Felder als das Muster besitzen, aber ansonsten übereinstimmen, ebenfalls matchen. Dies ist vergleichbar mit dem „Interface Inheritance“ wie es in Abschnitt 3.1.2 vorgestellt wurde. Es ist allerdings gegenüber dem Interface Inheritance bekannter Programmiersprachen eingeschränkt, da die Label der Tupel in der Reihenfolge identisch sein müssen, während z.B. in Java die Methoden eines Interfaces in einer beliebigen Reihenfolge definiert werden dürfen. Lumpe hat in seiner Arbeit (siehe [Lumpe 1999]) deswegen die Tupel durch „Formulare“ ersetzt, die als Abbildungen von den Labeln auf ihre Werte zu verstehen sind und deswegen den aufgezeigten Nachteil nicht aufweisen.

17.2.2 Syntax von Pict

Im Folgenden wird die Grammatik von Core Pict komplett in EBNF-Notation (siehe Abschnitt 16.2) angegeben und kurz erklärt.

Ein Prozess ist entweder ein leerer Prozess, eine Schreibaktion (`<WRITE>`), ein Prozess mit Leseaktion (`<READ>`), eine parallele Komposition, eine Deklaration oder eine If-Then-Else-Anweisung.

```
<PROC>        ::=  “()”
                |    <WRITE>
                |    <READ>
                |    <PARALLEL>
                |    <DECLARATION>
                |    <IFTHENELSE>.
```

Im Rahmen einer Schreibaktion wird auf einen Kanal <VAL$_1$> ein Wert[1] <VAL$_2$> geschrieben.

```
<WRITE>        ::=  <VAL1>!<VAL2>
```

Einer Leseaktion folgt immer ein weiterführender Prozess. Von einem Kanal <VAL> wird ein Wert gelesen, der mit dem Muster <PAT> (siehe unten) matcht. Danach wird mit dem Prozess <PROC> fortgefahren. Der Gültigkeitsbereich (Scope) der Variablen aus <PAT> ist auf <PROC> beschränkt[2].

```
<READ>         ::=  <VAL>?<PAT> = <PROC>
```

Zwei Prozesse <PROC$_1$> und <PROC$_2$> werden parallel ausgeführt, indem sie mit dem Parallelоperator „|“ verknüpft werden. Dieser Operator ist kommutativ und assoziativ.

```
<PARALLEL>     ::=  (<PROC1> "|"  <PROC2>)
```

Prozessen können Deklarationen vorangestellt werden. Es wird in <DEC> ein Kanal, eine rekursive Definition oder ein Typ neu definiert. Danach wird mit dem Prozess <PROC>, fortgefahren. Der Gültigkeitsbereich (Scope) der Variablen aus <DEC> ist auf <PROC> beschränkt.

```
<DECLARATION> ::=  "(" <DEC> <PROC> ")"
```

Die bedingte Ausführung wird wie auch in vielen anderen Programmiersprachen als If-Then-Else-Anweisung realisiert. Wenn der Wert <VAL> wahr ist, dann wird der Prozess <PROC1> sonst der Prozess <PROC2> ausgeführt.

```
<IFTHENELSE>   ::=  if <VAL> then <PROC1> else <PROC2>.
```

Ein Muster <PAT> ist entweder ein Variablenmuster oder ein Strukturmuster. Variablenmuster bestehen aus einer Variablen (<ID>), die von einem bestimmten Typ (<RTYPE>) sein muss. Strukturmuster bestehen aus einem Tupel von (evtl. durch <LAB> gekennzeichneten) Feldern, die wiederum mit Mustern belegt sein können.

---

[1] Anfangs werden in Pict wie auch im π-Kalkül als Werte nur Namen zugelassen. Später können darauf Kodierungen für natürliche Zahlen und andere Wertebereiche definiert werden. Auf die Herleitung wird hier verzichtet.

[2] Das Gleichheitszeichen („=“) zwischen PAT und PROC wirkt hier – wie auch bei Deklarationen (DEC) – etwas befremdlich, da keine Geichheitsbeziehung zwischen diesen beiden Teilen besteht, sondern eher eine Kausalbeziehung. Jedoch kann man diese Beziehung auch als Abstraktion verstehen: das Muster PAT abstrahiert vom weiteren Prozessverlauf PROC.

```
<PAT>          ::=  <ID> <RTYPE>
                |   “[” { <LAB> <PAT> } “]”.
```

Der Typ in einem Muster kann immer dann weggelassen werden (leeres Symbol `<empty>`), wenn er sich aus dem Zusammenhang ergibt. Ansonsten müssen auch die Variablen eines empfangenden Prozesses typisiert werden.

```
<RTYPE>        ::=  <empty>
                |   : <TYPE>.
```

Werte (`<VAL>`) sind entweder Konstanten, Pfade oder Strukturen. Es gibt mehrere elementare Datentypen und damit entsprechende Konstanten. In dieser Arbeit werden nur `true` und `false` vom Typ `Bool` genutzt. Werte eines Pfades lassen sich über Pfade (`<PATH>`) referenzieren. Komplexe Werte lassen sich aus Strukturen aufbauen.

```
<VAL>          ::=  <CONST>
                |   <PATH>
                |   “[” { <LAB> <VAL> } “]”.
```

Pfade sind entweder direkte Zugriffe auf Felder oder Zugriffe auf Felder hierarchischer Strukturen.

```
<PATH>         ::=  <ID>
                |   <PATH>“.”<ID>.
```

In dieser Arbeit werden nur zwei Konstante benötigt: `true` und `false`

```
<CONST>        ::= true | false.
```

Typen sind entweder typisierte Kanäle, boolesche Wahrheitswerte oder komplexe Datentypen. Ein komplexer Datentyp wird auf Basis von (evtl. durch Pfade bezeichnete) Tupeln definiert.

```
<TYPE>         ::=  ^<TYPE>
                |   Bool
                |   “[” { <LAB> <TYPE> } “]”     .
```

Deklarationen sind Kanaldefinition (Kanal `<ID>` vom Typ `<TYPE>`), Definitionen von Prozessagenten oder Typdefinitionen. Ein Prozessagent ist ein Prozess, der auf dem Kanal `<ID>` angesprochen wird, dessen Muster `<PAT>` die übergebenen Werte „matchen“ muss und der bei Aufruf den Prozess `<PROC>` ausführt. Typdefinitionen dienen der abkürzenden Schreibweise für einen (meist komplexen) Typ `<TYPE>`.

```
<DEC>          ::=  new <ID>:<TYPE>
                |   def <ID> <PAT> = <PROC>
                |   type <ID> = <TYPE>.
```

Ein Label ist entweder leer – dann besteht innerhalb einer Struktur nur über die Feldposition die Möglichkeit, auf Felder zuzugreifen – oder benennt ein Feld mit einem Bezeichner `<ID>`.

```
<LAB>          ::=  <empty>
                |    <ID> =    .
```

<ID> ist ein gültiger Identifizierer und <empty> ist das leere Wort.

17.2.3 Semantik von Pict

Die operationale Semantik von Core Pict wird in einer Reihe von Scoping-Regeln, die den Gültigkeitsbereich eines Kanalnamens definieren, und in Form zweier Relationen (Kongruenz- und Reduktionsrelation) angegeben. Die Scoping-Regeln definieren u.a., dass

1. der Name einer Deklaration (<DEC>) nur in dem Prozess <PROC> gebunden ist, der der Deklaration folgt, und

2. die durch ein Muster (<PAT>) eingeführten Variablen nur in dem Prozess gelten, der diesem Muster (nach dem „=“-Zeichen) folgt.

Die *Kongruenzrelation* besagt im Wesentlichen, dass die parallele Komposition („|“) kommutativ und assoziativ ist und der leere Prozess „()“ das Null-Element der parallelen Komposition ist, so dass (P|()) ≡ P gilt.

Die *Reduktionsrelation* definiert u.a. das Kommunikationsverhalten zwischen zwei Prozessen, wobei hier insbesondere die Asynchronität als Unterschied zum ursprünglichen π-Kalkül auffällt (vgl. Abschnitt 17.1), da nach der Schreibaktion kein weiterer Prozess definiert werden darf.

$$((x?y = P\ ) \mid x!z) \rightarrow P_{match(y,z)}$$

$P_{match(y,z)}$ ist der Prozess, der durch die Substitution von match(y,z) in P zustande kommt.

17.2.4 Syntactic Sugar

Es gibt einige Erweiterungen, die Pict syntaktisch an bekannte Programmiersprache annähert. So können spezielle Prozessagenten, die prinzipiell Funktionen darstellen, syntaktisch so definiert werden, dass sie auch wie Funktionen aussehen. Möchte man in Core-Pict eine Funktion, also einen Prozessagenten, der einen Wert zurück gibt, implementieren, so würde man normalerweise diesem Agenten einen Kanal (*Return-Kanal*) als Argument übergeben, auf dem er den Funktionswert zurückgibt. Im folgenden Beispiel wird ein Prozessagent nOt definiert, der einen booleschen Wert bekommt und auf einem ebenfalls übergebenen Antwortkanal die Negation dieses Wertes zurück liefert:

```
def nOt[x:Bool n:^Bool] =
        if x then n!false else n!true
```

Ein Aufruf dieser Funktion könnte wie folgt aussehen:

```
new ret:^Bool

run (nOt![true ret] | ret?bo = if bo then print!"1" else print!"2")
```

Um diese immer wieder auftretenden Strukturen explizit zu kennzeichnen, kann man die zur Rückgabe genutzten Return-Kanäle statt wie bisher mit dem Zeichen „^“ explizit mit dem Zeichen „/“ als einen solchen Kanal kennzeichnen:

```
def nOt[x:Bool n:/Bool] =
        if x then n!false else n!true
```

Der Aufruf eines so gekennzeichneten Prozessagenten (Funktion) darf dann in funktionaler, LISP-ähnlicher Schreibweise (siehe [Friedman und Felleisen 1989]) durchgeführt werden:

```
run if (nOt true) then print!"1" else print!"2"
```

Die Definition einer Funktion kann weiter vereinfacht werden, indem der Typ des Funktionswertes explizit nach hinten gestellt und der Funktionswert im Körper der Funktion direkt, d.h., ohne einen benannten Return-Kanal, angegeben wird.

```
def nOt(x:Bool): Bool =
        if x then false else true
```

Die Zuordnung von Werten zu Namen (Variablen) ist in Core Pict nur möglich, wenn man einen Wert auf einen Kanal schreibt und durch eine folgende Leseoperation einen Namen (bzw. eine Variable) mit dem zuvor geschriebenen Wert verbindet. Zur abkürzenden Schreibweise wurde in Pict der `val`-Befehl eingeführt, über den an einen Namen `x` ein Wert aus dem Funktionsaufruf `f` zugeordnet wird und der Wert von `f` im folgenden Prozess `proc` für `x` ersetzt wird.

```
(val x = f proc)
```

Diese syntaktischen Ergänzungen („syntactic sugar“) werden in Core Pict durch einen Vorübersetzer (pre-compiler) realisiert. In den nächsten Abschnitten wird von dieser vereinfachten Schreibweise Gebrauch gemacht.

17.2.5 Beispiel

In diesem Abschnitt wird ein kleines Pict-Beispiel präsentiert. Es wird ein Objekttyp definiert, der ein boolesches Attribut besitzt, das über eine Methode gesetzt und über eine weitere Methode abgefragt werden kann. Die Definition des Objekttyps besteht aus zwei Teilen: einer Definition eines Strukturtyps (`test`) und einer Verhaltensdefinition auf Grundlage eines Prozessagenten (`newTest`). Der Prozessagent `newTest` definiert einen Kanal vom Typ `^test`, auf dem Objekte vom Typ `test` angefordert werden können.

Der Strukturtyp `test` besitzt zwei Felder: `set` und `get`. Das `set`-Feld ist vom Typ `^[Bool]`, d.h. es bezeichnet einen Kanal auf dem Tupel mit einem Element vom Typ `Bool` gesendet werden können. Das `get`-Feld bezeichnet einen Kanal, auf dem Tupel geschickt werden können, die ihrerseits einen Kanal für den Typ `Bool` referenzieren.

```
type test = [
     set=^[Bool]
     get=^[^Bool]
]
```

Der Agent `newTest` macht im Wesentlichen zwei Dinge. Zum einen gibt er eine Struktur vom Typ `test` zurück, die dann die Schnittstelle eines Objekts (Instanz) darstellt. Zum anderen definiert er zwei interne Prozessagenten (`behaviour` und `initialized`), die das Verhalten dieses Objekts definieren. Für die Schnittstellen werden zwei interne Kanäle `s` und `g` definiert, die in Form von Methoden (`set` und `get`) angespochen werden können. Diese internen Kanäle werden dazu in einem Tupel des Typs `test` als Ergebnis beim Aufruf des Prozessagenten `newTest` nach außen hin bekannt gegeben.

Die internen Prozessagenten definieren zwei alternative Zustände, in denen sich das Objekt befinden kann. Der erste Zustand wird durch den Prozessagenten `behaviour` definiert. Dieser wartet so lange, bis auf dem Kanal `s` eine Nachricht zur Initialisierung des Objektes mit einem `Bool`-Wert eingegangen ist. Dann wechselt das Objekt den Zustand, indem es den Agenten `initialized` aufruft. In diesem Zustand (= Prozess) bleibt nun das Objekt für immer. Es können nun Nachrichten sowohl über den `set`- als auch über den `get`-Kanal geschickt werden. Eine `set`-Nachricht setzt den internen Wert des Objektes. Eine `get`-Nachricht erhält einen Kanal, auf dem der aktuelle Wert des Objektes zurückgegeben wird.

```
def newTest [t: ^test] =
(    new s:^[Bool]
     new g:^[^Bool]

     def initialized[b:Bool] =
          ( s?[x] = initialized![x]
          | g?[y] = (y!b | initialized![b]))

     def behaviour[] =
          s?[b] = initialized![b]

     run behaviour![]

     t![set = s get = g]
)
```

Um nun das ganze Programm zu testen, werden zwei Kanäle (`tr` und `br`) definiert. Der Kanal `tr` wird dem Prozessagenten `newTest` übergeben. Dieser schickt darauf einen Kanal `t`, über den ein neues Objekt referenziert werden kann. Über `t` können die internen Kanäle `s` und `g` über ihre externen Referenzen `set` und `get` aufgerufen werden. Der `set`-Kanal kann

direkt mit einem `Bool`-Wert aufgerufen werden. Dem `get`-Kanal wird eine Referenz auf den `Bool`-Kanal `br` mitgegeben, auf dem der aktuelle Wert des Objektes `t` zurückgegeben wird.

```
new tr:^test
new br:^Bool

run(
     newTest![tr]
 |   tr?t= (   t.set![false]
          |    t.get![br]
          |    br?b
               = if b then print!"True" else print!"False"
           )
)
```

17.2.6 Problem: Guarded Choice

Bei der Anwendung des obigen Beispiels gibt es ein Problem. Ist ein Objekt in dem Zustand `initialized`, kann nach dem Setzen eines Wertes durch `set` immer noch der alte Wert durch eine `get`-Nachricht zurückgegeben werden. Das liegt daran, dass der Agent `initialized` in seinem Zustand parallel sowohl eine `set`- als auch eine `get`-Nachricht anbietet und der Subprozess der `get`-Nachricht mit dem alten Wert bestehen bleibt, selbst wenn die `set`-Nachricht zu einem neuen Wert geführt hat. Richtig wäre es, alternativ nur eine der beiden Nachrichten zu verarbeiten. Es wird also ein Alternativoperator – oft als *Guarded Choice* bezeichnet – benötigt, wie er ursprünglich schon im synchronen π-Kalkül (siehe Abschnitt 17.1) eingeführt wurde. Dieser Operator gehört nicht zu den elementaren Operatoren der Sprache Core Pict, kann aber auf ihrer Grundlage nachgebildet werden. Die Idee besteht darin, die beiden alternativen Prozesse wie im Beispiel als parallele Prozesse zu implementieren, jedoch durch ein Synchronisationsflag dafür zu sorgen, dass, sobald eine Alternative ausgeführt wurde, die andere Alternative für immer gesperrt wird. Eine beispielhafte Implementierung ist in [Pierce und Turner 1994] angegeben. Dort wird der Choice-Operator durch folgendes Schema definiert[1]:

```
sync![
     (check
          (eventAction c₁ \PAT₁=PROC₁)
          (eventAction c₂ \PAT₂=PROC₂)
     )
]
```

[1] In der Originalarbeit von Pierce werden die Operatoren `check` und `eventAction` anders benannt und zudem als Infix-Operatoren präsentiert.

Die Pict-Funktion `eventAction` erhält zwei Argumente: einen Kanal c_i und einen anonymen Prozessagenten[1] $\PAT_i=\PROC_i$. Anonyme Prozessagenten sind Agenten ohne Namen, die direkt als Argumente übergeben werden können. Liegen Informationen auf einem Kanal c_i an, dann werden sie gelesen. Durch die Funktion `check` wird anhand eines Flags (`Lock`) geprüft, ob die andere parallel gestartete `eventAction`-Funktion schon vorher ausgeführt wurde. Ist das der Fall, so wird abgebrochen und die Daten werden unbenutzt auf den Kanal c_i zurückgeschrieben. Wurde noch keine andere Funktion ausgeführt, so wird der Prozessagent $\PAT_i=\PROC_i$ – er wird auch als *Aktion* oder *Aktionsagent* bezeichnet – gestartet. Die Funktion `check` hat bei diesem Vorgang nur die Aufgabe, die beiden `eventAction`-Aufrufe parallel zu starten und an beide Prozesse den Kanal für das Flag weiterzuleiten. Der Prozessagent `sync` stellt dieses Flag zur Verfügung und synchronisiert dadurch den gesamten Auswahlprozess.

Durch eine Erweiterung der in [Pierce und Turner 1994] vorgestellten Implementierung lässt sich die Ausführung der beiden Prozesse $\PROC_1$ bzw. $\PROC_2$ noch an zusätzliche Bedingungen knüpfen. Dazu werden weitere abstrakte Prozessagenten eingeführt, die eben der Beschreibung dieser Bedingungen dienen. Sie geben an, ob der jeweilige Prozess $\PROC_i$ grundsätzlich ausgeführt werden darf oder nicht.

Folgende allgemeine Schreibweise soll nun für diese im Sinne einer ECA-Regel (Event, Condition, Action) erweiterte Form des Guarded Choice eingeführt werden:

```
sync![
     (check
          (eventConditionAction #Type1 c1
                  \[PAT1 b: /Bool]=CPROC1 \PAT1=PROC1)
          (eventConditionAction #Type2 c2
                  \[PAT2 b: /Bool]=CPROC2 \PAT2=PROC2)
     )
]
```

Die Bedeutung dieses Konstruktes kann umgangssprachlich wie folgt formuliert werden: Auf den Kanälen c_1 und c_2 wird solange gewartet, bis auf einem Kanal ein Datum anliegt. In einem solchen Fall wird zuerst die zugehörige Bedingung ($\[PAT_i$ `b:/Bool]`$=\CPROC_i$) geprüft, und wenn sie erfüllt ist, wird der zugehörige Aktionsagent ($\PAT_i=\PROC_i$) ausgeführt. Außerdem wird die andere Alternative blockiert. Eine Bedingung wird genauso wie eine Aktion als Prozessagent definiert. Sie erhält als Argumente die gleichen Argumente

[1] Ein anonymer Prozessagent ist mit der Lambda-Abstraktion vergleichbar, wie sie aus dem Lambda-Kalkül (siehe [Church 1941]) oder der Programmiersprache LISP (siehe [Friedman und Felleisen 1989]) bekannt ist.

(PAT_i) wie die Aktion und zusätzlich einen Return-Kanal (`b`), auf dem das Ergebnis der Bedingungsauswertung (`true` oder `false`) zurückgegeben wird.

Ein Problempunkt wurde bislang der Einfachheit halber unterschlagen: Da die Sprache Core-Pict typisiert ist, bei der Entwicklung jedoch noch nicht klar ist, von welchem Typ die Kanäle c_1 und c_2 sein müssen, wird hier noch eine weitere, bislang nicht erwähnte Eigenschaft von Pict genutzt: die Parametrierung durch Typen. Ein Typ kann zur Laufzeit eines Programms als Parameter an einen Prozess übergeben werden, wodurch die Typen der Argumente eines Prozesses erst beim Aufruf festgelegt werden. In dem dargestellten Schema sieht man, dass Typen durch den Präfix „#“ gekennzeichnet werden. In der `eventConditionAction`-Funktion wird ein Typ $Type_i$ übergeben. Er definiert den Typ des Kanals c_i und der Argumente der Aktion sowie des ersten Argumentes (PAT_i) der Bedingung.

17.2.7 Erweitertes Guarded Choice

In diesem Abschnitt wird eine im Rahmen dieser Arbeit entwickelte Formalisierung des erweiterten Guarded Choice gemäß des vorgestellten *ECA-Musters* in Pict angegeben. Dazu werden die drei in Abschnitt 17.2.6 vorgestellten Prozessagenten bzw. Funktionen definiert.

Zuerst wird ein Typ von Prozessagenten (`lockFun`) definiert, denen ein Argument übergeben werden muss, nämlich der Name eines Kanals, auf dem die Namen von Kanälen für `Bool`-Werte verschickt werden können. Kurz gesagt: Ein Prozessagent vom Typ `lockFun` bekommt als einziges Argument einen Kanal für Kanäle für `Bool`-Werte übergeben.

```
type lockFun =  /[^^Bool]
```

Nun wird der Prozessagent `sync` definiert, der als Argument eine Funktion `e` vom Typ `lockFun` erhält. Der Agent `sync` erzeugt zuerst ein Flag `lock` vom Typ „Kanal für Kanäle vom Typ `Bool`“ und zwei lokale Prozessagenten: `lockFalse` und `lockTrue`. Diese beiden Agenten definieren das Verhalten von `sync`: Auf die erste Anfrage über den Kanal `lock` wird auf dem mitgelieferten Antwortkanal `r` der Wert `true`, bei allen weiteren Anfragen der Wert `false` zurückgeschickt. Dadurch wird sichergestellt, dass im Sinne der gewünschten Auswahl nur genau eine Alternative ausgesucht wird. Das Verhalten der Agenten wird durch den `run`-Befehl initialisiert, und danach wird die übergebene Funktion `e` mit dem Flag `lock` aufgerufen.

```
def sync[e: lockFun] =
(       new lock: ^^Bool
        def lockFalse[] = lock?r = (r!false | lockFalse![])
        def lockTrue[]  = lock?r = (r!true  | lockFalse![])
        run lockTrue![]
        e![lock]
)
```

Die Pict-Funktion `check` erhält als Argumente zwei Funktionen vom Typ `lockFun` und erzeugt daraus eine neue Funktion vom gleichen Typ. Eine solche neue Funktion erhält das schon erwähnte Flag `lock` als Argument und ruft damit die beiden übergebenen Funktionen auf:

```
def check (e1: lockFun   e2: lockFun) : lockFun =
        \[lock: ^^Bool] = (e1![lock] | e2![lock])
```

Die letzte benötigte Funktion `eventCondAction` codiert die Schaltlogik einer ECA-Regel in einer Funktion vom Typ `lockFun`, die dann als Funktionswert zurückgegeben wird. Dazu werden ihr vier Argumente übergeben: ein Typ `X`, ein Kanal `c` für Daten des Typs `X`, eine boolesche Funktion `cond` und ein Prozessagent `action`. Die erzeugte Funktion vom Typ `lockFun` liest, wenn sie aufgerufen wird, zunächst Daten vom Kanal `c`. Danach wird geprüft, ob diese Daten der Bedingung der Funktion `cond` genügen. Ist das der Fall, wird anhand des Flags `lock` getestet, ob der Prozessagent `action` noch ausgeführt werden darf oder nicht. Darf er ausgeführt werden, so werden ihm die aus dem Kanal `c` gelesenen Daten übergeben. In allen anderen Fällen werden die gelesenen Daten zurück auf den Kanal `c` geschrieben:

```
def eventCondAction (#X c:^X  cond:/[X /Bool] action:/X) : lockFun =
    \[lock: ^^Bool] =
        c?v =
        if (cond v) then
                ( new rb: ^Bool
                  (lock!rb | rb?b = if b then action!v else c!v)
                )
        else    c!v
```

Nun kann die Implementierung aus dem Beispiel in Abschnitt 17.2.5 korrigiert werden. Dazu wird im Prozessagenten `initialized` die parallele Ausführung der beiden Leseaktionen durch das hier vorgestellte Schema ersetzt. Der Prozessagent hat dann folgendes Aussehen:

```
def initialized[b:Bool] =
sync![
(check   (eventCondAction #[Bool]
                        s
                \[y: [Bool] x:/Bool]  = x!true
                \[y: Bool]  = initialized![y])
         (eventCondAction #[^Bool]
                g
                \[y: [^Bool] x:/Bool]  = x!true
                \[z: ^Bool] = (z!b | initialized![b]))
)
]
```

Es ist anzumerken, dass die beiden Funktionen für die Bedingungen in diesem Beispiel immer erfüllt sind, da sie ausschließlich den Wert `true` zurückgeben.

Das hier vorgestellte ECA-Schema wird zur Definition der Semantik von Zustands/Transitionsbeschreibungen von SCDL in Abschnitt 7.3 benötigt.

18 Beschreibungslogik

Die terminologischen Repräsentationsformalismen (sogenannte *Beschreibungslogiken*, siehe [Baader et al. 1992], [Bibel 1993]) sind historisch als Folge der Forderung nach einer formalen Semantik von semantischen Netzen und Frames entstanden. Diese Zielsetzung hat zu einer weitgehenden logischen Formalisierung terminologischer Systeme geführt. Weiterhin hat man sich um eine Vergrößerung der Ausdruckskraft dieser Systeme bemüht. Damit jedoch Schlussfolgerungen in terminologischen Logiken entscheidbar bleiben, muss die Ausdruckskraft kleiner sein als die der Prädikatenlogik (siehe [Schöning 1987]). Deshalb sind in terminologischen Logiken nur variablenfreie Terme zugelassen. Eine weitere Eigenschaft dieser Logiken ist die getrennte Beschreibung der im Problembereich relevanten Begriffe in einer Terminologie (TBox) und die Beschreibung der Objekte und ihrer Verbindungen zu den Begriffen in einer Weltbeschreibung (ABox). Terminologische Logiken besitzen damit die Vorteile der logischen Formalismen, insbesondere eine formale Semantik und formale Methoden zum Schlussfolgern. Die erste terminologische Sprache war KL-ONE (siehe [Brachmann und Schmolze 1985]). KL-ONE übt bis heute einen starken Einfluss auf die Entwicklung von Wissensrepräsentationssprachen aus. Eine wichtige Inferenzfähigkeit von terminologischen Systemen ist die Berechnung von Subsumtionsbeziehungen zwischen Konzepten, d. h. die Ober-Unterbegriff-Beziehungen, die durch die Konzeptdefinitionen induziert werden. Die verschiedenen terminologischen Systeme unterscheiden sich z. B. hinsichtlich der Effizienz der Berechnung der Subsumtionsrelation. Für eine vollständige Berechnung wie in KRIS (siehe [Baader und Hollunder 1991]) sind NP-vollständige Algorithmen mit im „worst case“ exponentiellem Laufzeitverhalten notwendig, während z. B. in Systemen aus der CLASSIC-Familie (siehe [Borgida et al. 1989]) die Sprache weiter eingeschränkt wird, so dass Algorithmen mit polynomialer Laufzeit existieren. Es wird zur Zeit offensichtlich noch der richtige Kompromiss zwischen Ausdrucksfähigkeit und Komplexität gesucht.

18.1 Der terminologische Formalismus

In diesem Abschnitt werden die in verschiedenen terminologischen Systemen vorhandenen Sprachkonstrukte mit ihrer Syntax und Semantik eingeführt, um einen Einblick in die Ausdrucksfähigkeit von terminologischen Systemen zu geben. Dazu soll die Sprache $\mathcal{ALC}$ (siehe [Schmidt-Schauß und Smolka 1991]) dienen. Diese Sprache enthält den minimalen Sprachumfang einer terminologischen Logik und kann damit als Prototyp für die Definition von mächtigeren Sprachen angesehen werden. Die ursprünglich in KL-ONE verwendete Sprache ist zwar ausdrucksmächtiger als die hier vorgestellte Sprache, dafür sind wichtige Inferenzprobleme in KL-ONE unentscheidbar. In $\mathcal{ALC}$ sind diese Probleme in abgeschwächter Form jedoch entscheidbar, so dass ein korrekter und vollständiger Subsumtionsalgorithmus für $\mathcal{ALC}$ angegeben werden kann.

Daten- und Wissensbasen der Informatik können als Beschreibungen von Ausschnitten der „Wirklichkeit“ (*Diskurse*) verstanden werden. Solche Beschreibungen basieren oft auf individuellen Objekten, die über Beziehungen miteinander verbunden sind und nach verschiedenen Eigenschaften in Klassen gruppiert werden können. *Beschreibungslogiken* (*Description Logics*, DLs), oft auch als *terminologische Logiken* bezeichnet, sind eine Klasse von Sprachen, die zur Beschreibung und zum Arbeiten mit solchen Beschreibungen entwickelt wurden. Ihre Besonderheit liegt darin, dass die Klassen von Objekten – in DLs werden sie als Begriffe[1] (*Concepts*) bezeichnet – intensional beschrieben werden können, also mittels Beschreibungen, die auf die *Eigenschaften* abzielen, die ein Objekt haben muss, damit es unter diesen Begriff fällt. Diese Beschreibungen basieren auf Sprachen, die es erlauben, komplexere Beschreibungen systematisch auf einfachere Beschreibungen zurückzuführen. Dazu gehören u.a. Einschränkungen, die die binären Beziehungen (*Roles*) zwischen den Objekten verschiedener Konzepte betreffen.

Ein einfaches Beispiel wird durch folgende Beschreibung gegeben:

$$\text{SPIEL} \sqcap \geq 4\ \text{teilnehmer} \sqcap \forall\ \text{teilnehmer:}(\text{PERSON} \sqcap \forall\ \text{geschlecht:WEIBLICH})$$

Hier wird ein Spiel beschrieben, das mehr als vier Teilnehmer benötigt. Alle Teilnehmer müssen Personen und weiblichen Geschlechts sein.

Intensional betrachtet, handelt es sich bei dem Zeichen $\sqcap$ um eine konjunktive Verknüpfung der Eigenschaften. Aus extensionaler Sicht, also aus Sicht der Objekte, die diese Eigenschaften haben, handelt es sich um eine mengenorientierte Durchschnittsoperation, wie im nächsten Abschnitt noch genauer gezeigt wird.

Die Beschreibung teilt sich in drei Teile. „SPIEL“ ist ein atomares Konzept, dass nicht weiter definiert wird. Durch den zweiten Ausdruck werden alle Objekte ermittelt, die eine Rolle (auch: Beziehungstyp) „teilnehmer“ besitzen, und die über mehr als drei konkrete Beziehungen dieses Typs zu Objekten anderer Konzepte („Füller“) verbunden sind. Im dritten Teil ist beschrieben, dass zusätzlich für alle diese Füller gelten muss, dass sie Personen sind und „alle ihre Geschlechter“ weiblich sind. In der Realität hat eine Person natürlich nur ein Geschlecht. Dieser Sachverhalt wird hier jedoch nicht gefordert.

Neben dieser intensionalen Beschreibung – sie wird in terminologischen Sprachen auch als *TBox* (terminologische Box) bezeichnet –, können auch konkrete Objekte (Instanzen)

---

[1] Statt des Wortes „Begriff“ wird im Folgenden auch „Konzept“ genommen, da er dem englischem Wort „Concept“ zumindest lexikographisch näher kommt. Es soll hier explizit auf die gleiche Bedeutung des englischen Wortes „Concept“ und des deutschen Wortes „Begriff“ hingewiesen werden.

beschrieben werden. Diese werden in der sogenannten *ABox* (assertionale Box) beschrieben. Während also die TBox Begriffe repräsentiert, werden in der ABox konkrete Aussagen über Tatsachen der Diskurswelt gemacht.

Grundsätzlich eignen sich Beschreibungen aus Sicht der automatisierten Verarbeitung z.B. mittels eines wissensbasierten Systems für folgende Zwecke:

- Konzepte können auf der Grundlage von primitiven und anderen (schon definierten) Konzepten induktiv beschrieben werden. Zwischen verschiedenen Beschreibungen der Konzepte lassen sich zudem Subordinationsbeziehungen (Ober-Unterbegriff-Beziehungen) automatisch ermitteln. (Definierte) Konzepte lassen sich dazu in eine Halbordnung einordnen (klassifizieren). Diese Halbordnung basiert auf der sogenannten Subsumtionsrelation (siehe Abschnitt 18.3). Sie lässt sich insofern für diese Arbeit nutzen, als dass abstrakte Konfigurationen immer eine Menge weniger abstrakter oder sogar gebundener Konfigurationen einschließen (subsumieren).
- Beschreibungen können für Anfragen genutzt werden, um z.B. alle Objekte, die einer Beschreibung genügen, aus einer Wissensbasis zu ermitteln.
- Beschreibungen können widersprüchlich sein. Widersprüchliche Beschreibungen können dann durch entsprechende DL-Systeme automatisch abgelehnt werden. Dieses ist eine für diese Arbeit wesentliche Eigenschaft. Konfigurationsoperationen, wie sie in Abschnitt 9.2.4 vorgestellt werden, können, wenn sie in zu einer inkonsistenten Konfiguration führen würden, von vornherein abgelehnt werden. Die Konsistenz einer Konfiguration bleibt dadurch automatisch gewahrt.

18.2 Abstrakte Syntax von Beschreibungslogiken

Es gibt eine Reihe verschiedener Beschreibungslogiken. In diesem Abschnitt wird neben der allgemeinen Beschreibungssprache $\mathcal{ALC}$ die dem System NEOCLASSIC zugrunde liegende Beschreibungslogik genauer vorgestellt. NEOCLASSIC wurde gewählt, weil es sich als sehr performantes System für Konfigurationsprobleme (siehe [McGuinness und Wright 1998]) etabliert hat. Ferner wurde in der Arbeit von [Jaekel 1999] gezeigt, dass es sich auch für die Konfiguration framework-basierter Softwaresysteme sehr gut eignet.

Die Syntax wird zuerst in ihrer abstrakten Struktur in Anlehnung an $\mathcal{ALC}$ beschrieben. Sie ähnelt der Syntax, wie sie auch in den meisten theoretischen Arbeiten zum Thema (für eine Übersicht siehe z.B. [Baader et al. 1992]) genutzt wird. In der folgenden Tabelle ist neben der abstrakten Syntax auch die Syntax von NEOCLASSIC aufgeführt. Einige Standardoperatoren wie die Disjunktion oder die Negation wurden in NEOCLASSIC aus Effizienzgründen nicht implementiert. Dadurch konnte der Subsumtionsalgorithmus wesentlich beschleunigt werden (siehe [Borgida und Patel-Schneider 1994]).

18.2.1 TBox-Syntax

Zuerst wird die Syntax zur Beschreibung der TBox (begriffliche Ebene) vorgestellt. Einfache Beschreibungen können dazu gemäß der Konstrukte aus Tabelle 18.1 induktiv aufgebaut werden.

Die Symbole, die auch in den folgenden Tabellen benutzt werden, haben folgende Bedeutung: S steht für ein neu eingeführtes Symbol, das als Abkürzung für eine Konzeptbeschreibung genutzt werden kann. Q steht für eine Rollendefinition. E, C und D stehen für Konzeptbeschreibungen, R für Rollen, A und B für Attribute sowie n für eine natürliche, positive Zahl. Ausgangspunkt aller Konzepte ist das generellste Konzept, das TOP-Element. Dieses Konzept umfasst (subsumiert) alle anderen Konzepte. Im Gegensatz dazu steht das Bottom-Element, das kein anderes Konzept subsumiert und die Inkonsistenz symbolisiert.

Abstrakte Syntax	Erklärung	**NEOCLASSIC**
TOP (auch: $\top$)	Top-Element	classic-thing
Bottom (auch: $\bot$)	Bottom-Element	keine direkte Entsprechung in NEOCLASSIC*
E	atomarer (undefinierter) Konzeptname	`[A-Za-z]+[A-Za-z_]*`
$C \sqcap D$	Durchschnitt (Konjunktion von Konzepten)	`(and C D)`
$C \sqcup D$	Disjunktion von Konzepten	keine direkte Entsprechung in NEOCLASSIC**
$\forall$ R:C	Restriktion der Werte einer Rolle	`(all R C)`
$\forall$ A:C	Restriktion der Werte eines Attributs	`(all A C)`
$\geq$ n R	Minimum der Anzahl der Rollenfüller	`(atLeast n)`
$\leq$ n R	Maximum der Anzahl der Rollenfüller	`(atMost n)`
$A_1 \circ ... \circ A_k \downarrow B_1 \circ ... \circ B_h$	Gleichheit von Attributen (Co-Referenz)	keine direkte Entsprechung in NEOCLASSIC***
*) Ein Bottom-Element steht für eine inkonsistente Beschreibung. Solche Beschreibungen werden in NEOCLASSIC abgelehnt und durch eine Fehlermeldung angezeigt.		
**) Die Disjunktion kann durch Regeln (Rules) in NEOCLASSIC nachgebildet werden, die jedoch dann einen eher imperativen Charakter hat als die ursprüngliche Disjunktion.		
***) Die Gleichheit von Attributen kann in NEOCLASSIC durch Regeln (Rules) nachgebildet werden.		

Tabelle 18.1: Syntax für Konzeptbeschreibungen

Mittels terminologischer Axiome können symbolische Bezeichner von Konzepten und Rollen eingeführt werden, die im Wesentlichen der abkürzenden Schreibweise dienen. In der Tabelle 18.2 sind diese Axiome aufgeführt.

In den meisten terminologischen Logiken wie auch in NEOCLASSIC dürfen keine zyklischen Definitionen innerhalb einer TBox definiert werden.

Abstrakte Syntax	Erklärung	NEOCLASSIC
S=C	Definition	`(createConcept S C)`
S⊑C	Primitive Definition	`(createConcept S C true)`
Q=R	Rollendefinition	`(createRole Q)*`
A=B	Attributdefinition	`(createRole A true)**`
*) In NEOCLASSIC werden die Konzepte R der an einer Rolle beteiligten Individuen nicht vorgegeben.		
**) Wie in Abschnitt 18.2.1 zu sehen ist, können in NEOCLASSIC Attribute nur als spezielle Rollen, mit der Kardinalität 0..1 dargestellt werden.		

Tabelle 18.2: Terminologische Axiome

18.2.2 ABox-Syntax

Mit Hilfe von Assertionen können die in der TBox eingeführten Konzepte und Rollen (inklusive Attribute) mit konkreten Objekten (Individuen) gefüllt werden. Dazu werden sogenannte assertionale Axiome zur Verfügung gestellt, die in Tabelle 18.3 aufgeführt werden. Dabei repräsentieren a und b Individuen, C eine Konzeptbeschreibung und R einen Rollenterm.

Abstrakte Syntax	Erklärung	NEOCLASSIC
a:C	Definition eines Individuums a des Konzeptes C	`(createIndividual a C)`
a R b	Definition einer Beziehung vom Typ R zwischen den Individuen a und b.	`(addToldInformation a (fills R b))`

Tabelle 18.3: Assertionale Axiome

18.3 Semantik von Beschreibungslogiken

Die (denotationale) Semantik von Beschreibungslogiken wird i.d.R. rekursiv auf den möglichen Extensionen, also konkreten Ausprägungen eines Konzeptes definiert, die den Konzepten durch eine Interpretation *I* zugeordnet werden. In der folgenden Definition wird nebenbei schon auf eine Besonderheit von NEOCLASSIC, nämlich den Host-Bereich eingegangen. In diesen Bereich fallen die systemseitig vordefinierten Wertebereiche wie ganze Zahlen (Integer) und Zeichenketten (Strings).

Definition:

Eine Interpretation *I* besteht aus einer Domäne Δ und einer Interpretationsfunktion $.^I$. Die Domäne ist in zwei disjunkte Mengen Δ_C (Classic-Bereich) und Δ_H (Host-Bereich) unterteilt. In Δ_H sind systemseitig vorgegebene Konzepte wie STRING und INTEGER definiert. Die Interpretationsfunktion ordnet den atomaren Konzepten wie folgt Extensionen aus Δ zu:

- Die Extensionen eines atomaren Konzeptes E ist eine Teilmenge E^I von Δ_C.
- Die Extensionen einer Rolle R ist eine Teilmenge R^I von Δ_C x Δ.
- Die Extensionen eines Attributs A ist eine totale Funktion A^I von Δ_C nach Δ.

Die Extensionen C^I einer nicht-atomaren NEOCLASSIC-Beschreibung können nun wie folgt berechnet werden:

- $TOP^I = \Delta_C$.
- $(C \sqcap D)^I = C^I \cap D^I$.
- $(C \sqcup D)^I = C^I \cup D^I$.
- $(\forall\ P{:}C)^I = \{\ d \in \Delta_C \mid \forall x : (d,x) \in P^I \Rightarrow x \in C^I\ \}$. Hierdurch werden alle Extensionen bestimmt, die in Δ_C liegen und deren Rollen- bzw. Attribut-Extensionen („Füller“) P Elemente von C sind. Das „P“ steht hier stellvertretend für ein „R“ (bei Rollen) oder „A“ (bei Attributen).
- $(\geq n\ R)^I$ (bzw. $(\leq n\ R)^I$) = sind jene Extensionen aus Δ_C mit mindestens (bzw. höchstens) n Extensionen („Füllern“) für die Rolle R.
- $(\ A_1 \circ ... \circ A_k \downarrow B_1 \circ ... \circ B_h\ )^I = \{\ d \in \Delta_C \mid A_k^I\ (...A_1^I\ (d)...) = B_h^I\ (...B_1^I\ (d)...)\}$. Hierdurch werden alle Extensionen aus Δ_C ausgewählt, deren Extensionen der beiden Attribute, die über die Attributpfade $A_1 \circ ... \circ A_k$ und $B_1 \circ ... \circ B_h$ bezeichnet werden, identisch sind.

In der Möglichkeit zur Berechnung der Subsumtionsbeziehung zwischen verschiedenen Konzepten bzw. ihren Beschreibungen liegt die wesentliche Stärke von DL-Systemen. Sie berechnen, wie schon oben angedeutet, ob eine Beschreibung genereller als eine andere ist, oder anders ausgedrückt, ob eine (abstrakte) Beschreibung eine andere (konkretere) Beschreibung subsumiert.

Eine Beschreibung D_1 *subsumiert* eine andere Beschreibung D_2, wenn für alle Interpretationen I gilt, dass $D_2^I \subseteq D_1^I$.

Glossar

Akteur

Ein Akteur ist ein Teilnehmer am Entwicklungsprozess (→Makroprozess) komponentenbasierter →Anwendungssysteme. Akteure sind meist Unternehmen (z.B. Softwarehersteller, Beratungsunternehmen oder anwendende Unternehmen) oder Organisationseinheiten (z.B. die DV-Abteilung) innerhalb eines Unternehmens, die in unterschiedlichen →Rollen im Makroprozess auftreten können.

Anwender

Ein Anwender ist eine →Rolle im →Makroprozess. Anwender sind →Akteure (Unternehmen, Abteilung), die ein →Anwendungssystem in einem Aufgabenbereich (= Anwendungsbereich) einsetzten. Ein Anwender hat i.d.R. mehrere Mitarbeiter, die als Benutzer des Anwendungssystems auftreten.

Anwendungsarchitekt

Anwendungsarchitekt ist eine →Rolle im →Makroprozess. Ein Anwendungsarchitekt ist ein →Akteur, der aus atomaren und/oder komplexen →S-Komponenten größere, komplexe Komponenten entwickelt, die als eigenständige Produkte (→Anwendungssoftwareprodukte) verkauft bzw. lizenziert werden können.

Anwendungssoftwareprodukt

Ein Anwendungssoftwareprodukt ist eine komplexe →S-Komponente, die vollständig ist, d.h. ohne weitere S-Komponenten eingesetzt werden kann. I.d.R. muss ein Anwendungssoftwareprodukt konfiguriert werden, um es an die konkreten Bedürfnisse eines →Anwenders anzupassen. Ein vollständig konfiguriertes Anwendungssoftwareprodukt wird als →Anwendungssystem bezeichnet.

Anwendungssystem

Unter einem Anwendungssystem wird die einsatzbereite →Konfiguration (Installation) eines →Anwendungssoftwareprodukts verstanden. Insbesondere wurden alle benötigten Konfigurationseinstellungen an diesem Produkt vorgenommen, damit es weitestgehend die Anforderungen des Anwenders abdeckt.

ARIS-Methode

ARIS (Architektur integrierter Informationssysteme) ist eine Familie von Diagrammsprachen, die zur Beschreibung der Aufbau- und Ablauforganisation sowie der Informationsstrukturen eines Unternehmens dient. ARIS wird vor allem zur →Geschäftsprozessmodellierung im Rahmen von Business-Process-Reengineering-Maßnamen sowie zur Anforderungsanalyse

und Sollkonzeption betrieblicher →Anwendungssysteme eingesetzt. Die →Ereignisgesteuerte Prozesskette ist eine Teilsprache von ARIS.

Definitionszeit (eines Anwendungssystems)

Die Definitionszeit ist der Zeitraum, in dem ein →Anwendungssystem entwickelt wird. In diesen Zeitraum fallen die Entwicklung der →Komponenten (→Deklarationszeit), die Entwicklung eines →Anwendungssoftwareprodukts (→Konstruktionszeit) und die Konfiguration eines Anwendungssoftwareprodukts zu einem Anwendungssystem (→Konfigurationszeit). An die Definitionszeit schließt sich die →Laufzeit des Anwendungssystems an.

Deklarationszeit

Die Deklarationszeit ist der Zeitraum innerhalb der →Definitionszeit eines →Anwendungssystems, in dem die Bestandteile (→Klassen, →Signaturen) einer Komponente definiert und bezeichnet werden. Die entsprechenden Bezeichner werden durch den →Komponentenhersteller festgelegt und werden im weiteren Entwicklungsprozess (→Makroprozess) zum Aufbau eines Anwendungssystems referenziert.

Entwurfsmuster

Entwurfsmuster (Design Patterns) sind in der objektorientierten Softwareentwicklung etablierte, musterhafte Beschreibungen von typischen Strukturen und Verhaltensweisen mehrerer Objekte. Sie sind im Vergleich zu →Frameworks kleinere architektonische Einheiten (engl. Microarchitectures), die als Designrichtlinien für den Aufbau von Frameworks und framework-basierten →Anwendungssystemen genutzt werden können.

Ereignisgesteuerte Prozesskette

Die Ereignisgesteuerte Prozesskette (EPK) ist eine Teilsprache der Sprachfamilie der →ARIS-Methode. Sie dient zur Beschreibung von →Geschäftsprozessen (→Geschäftsprozessmodellierung).

Ereignis

Ein Ereignis ist ein Schema (oder eine Typisierung) eines →Geschehnisses. Um ein Ereignis von einem anderen Ereignis abzugrenzen, definiert man den Zeitraum zwischen diesen Ereignissen oft als →Zustand. In der →Ereignisgesteuerten Prozesskette werden Zustände als Ereignisse bezeichnet.

Framework

Ein Framework (Rahmenwerk) ist – sehr allgemein gesprochen – ein Softwaresystem mit verbindlichem Design und zugehöriger Implementierung, in das Softwareelemente (→Klassen, →Komponenten) an dafür vorgesehenen Stellen (Hot Spots, →Slots) eingesetzt

werden können, um ein vollständiges →Anwendungssoftwareprodukt zu konstruieren. Müssen dazu individuell →Klassen entwickelt werden, so spricht man von einem →Klassen-Framework. Brauchen hingegen nur fertige Komponenten eingesetzt werden, so bezeichnet man den Framework als →Komponenten-Framework.

Funktionalität

Unter der Funktionalität einer →Komponente wird die Menge aller →Interaktionen verstanden, die mit ihr realisierbar ist. Sie wird durch die Spezifikation des →aufrufbaren Verhaltens einer Komponente vorgegeben. Wenn die Menge der Interaktionen durch →Parametrierung, also Parameterwerte beeinflusst, d.h. eingeschränkt werden kann, so wird diese eingeschränkte Menge als →Funktionsweise bezeichnet.

Funktionsweise

Unter der Funktionsweise einer →Komponente versteht man den Teil der →Funktionalität einer Komponente, der durch →Parametrierung der Komponente aktiviert wurde. Eine Komponente besitzt je nach Parametrierungsmöglichkeiten unterschiedliche Funktionsweisen. Alle Funktionsweisen zusammen ergeben dann die →Funktionalität der Komponente.

Geschäftsprozess

Unter einem Geschäftsprozess wird eine Abfolge von →Interaktionen zwischen mehreren Objekten (Menschen, Maschinen, →Anwendungssystemen) verstanden, die einem betrieblichen Ziel dienen. Bei der Durchführung der (Inter-)Aktionen werden weitere Objekte verbraucht (Input), erzeugt (Output) und transformiert (In- und Output). Diese Objekte können Materialien, Produkte, Informationen und allgemein Dienstleistungen sein. Im Kontext des →R/3-Referenzmodells werden unter systemnahen Geschäftsprozessen Beschreibungen von Transaktionsfolgen verstanden, die einer betriebswirtschaftlichen Aufgabenstellung dienen.

Geschäftsprozessmodellierung

Unter der Geschäftsprozessmodellierung wird das Beschreiben von Geschäftsprozessen mittels mehr oder weniger formalen Sprachen (z.B. der →Ereignisgesteuerten Prozesskette) verstanden. Das Resultat ist ein →Geschäftsprozessmodell.

Geschäftsprozessmodell

Ein Geschäftsprozessmodell ist die Beschreibung einer Menge ähnlicher →Geschäftsprozesse. Dabei kann es sich um vergangene oder für die Zukunft geplante Geschäftsprozesse handeln. Geschäftsprozessmodelle werden oft mittels graphischer Diagrammsprachen (z.B. der →Ereignisgesteuerten Prozesskette) dargestellt.

Geschehnis

Ein Geschehnis ist ein Gegenstand, der im Gegensatz zu einem „Ding“ zeitlich begrenzt ist. Unter Geschehnissen werden z.B. →Handlungen, Regungen und Bewegungen verstanden.

Handlung

Handlungen sind durch den Handelnden veranlasste und beabsichtigte →Geschehnisse.

Hot Spot

siehe →Framework.

Informationssystem, betriebliches

Ein betriebliches Informationssystem ist das gesamte informationsverarbeitende Teilsystem eines Unternehmens. Ein betriebliches Informationssystem ist ein sozio-technisches System, da es sich sowohl aus den Mitarbeitern des Unternehmens als auch aus den →betrieblichen Anwendungssystemen zusammensetzt.

Interaktion

Interaktionen sind partnerorientierte →Handlungen.

K-Komponente

Eine K-Komponente ist eine →Komponente im Sinne der Konfiguration. In Konfigurationsproblemen werden im Wesentlichen elementare (atomare) Komponenten und komplexe Komponenten, die aus anderen Komponenten zusammengesetzt werden, unterschieden.

Klasse

Eine Klasse ist ein Beschreibung, die den Zustand (in Form von Attributen) und das →Verhalten von Objekten definiert. Objekte sind Instanzen einer Klasse. Eine Klasse gehört immer genau zu einer →S-Komponente.

Klassen-Framework

Ein Klassen-Framework – auch objektorientierter Framework genannt – besteht aus einer Menge zusammenarbeitender (kooperierender) →Klassen, von denen einige abstrakt sind. Zu den abstrakten Klassen müssen von dem Entwickler, der den Framework nutzen möchte, konkrete Klassen implementiert werden. Die abstrakten Klassen bilden also die Ansatzpunkte (→Hot Spot) für die Wiederverwendung der Frameworks. Durch den Framework ist die Architektur und damit der Softwareentwurf vorgegeben, an dem die zusätzlich zu entwickelnden Klassen ausgerichtet werden müssen.

Kommunikation

Kommunikationen sind →Interaktionen, die auf symbolischer (verbaler, schriftlicher usw.) Ebene stattfinden.

komplexe Komponente

Siehe →Komponente.

Komponente

Der Begriff Komponente wird in dieser Arbeit in zwei verschiedenen Bedeutungen benutzt: als Softwarekomponente (→S-Komponente) und als Komponente im Sinne der Konfiguration (→K-Komponente). Eine Komponente kann aus mehreren anderen Komponenten zusammengesetzt werden und wird dann als komplexe Komponente bezeichnet.

Komponenten-Framework

Ein Komponenten-Framework ist eine →S-Komponente, die das Zusammenspiel verschiedener anderer (potentieller) Komponenten durch →Kontrakte vorgibt. Die möglichen anderen Komponenten müssen in vorgegebene Steckplätze (Slots) eingesteckt werden, damit der Framework tatsächlich genutzt werden kann. Die Slots geben →Kontrakte vor, die die jeweiligen Komponenten einhalten müssen.

Komponentenhersteller

Ein Komponentenhersteller ist eine →Rolle im →Makroprozess. Komponentenhersteller sind →Akteure, die atomare →S-Komponenten entwickeln und an →Anwendungsarchitekten verkaufen bzw. lizenzieren.

Komponenteninstanz

Eine →S-Komponente ist ein Element der Softwareentwicklung, das erst durch Einbinden in eine konkrete Laufzeitumgebung (→Laufzeit) genutzt werden kann. Eine solche eingebundene Komponente wird dann auch als Komponenteninstanz bezeichnet. Innerhalb einer Laufzeitumgebung sollte es aus Effizienz- und administrativen Gründen – um z.B. zur Laufzeit den Release-Wechsel einer Komponente zu vereinfachen – möglichst nur eine Komponenteninstanz geben.

Komposition

Unter Komposition wird die Zusammensetzung (bzw. das Komponieren) mehrerer →Komponenten zu größeren (komplexen) Komponenten verstanden. Die Komposition kann z.B. durch →Parametrierung oder durch die Anwendung von Paralleloperatoren auf mehrere Komponenten (parallele Komposition) vollzogen werden. Die Komposition kann von Hand oder auch auf Basis von Konfigurationssystemen vorgenommen werden. Bei der

Konfiguration kann aus einer gebundenen →Konfiguration als Ergebnis eine Komposition abgeleitet werden. Das Ergebnis der Komposition wird als Kompositum bezeichnet.

Kompositum

Siehe →Komposition.

Konfiguration

Das Wort „Konfiguration“ ist ein Homonym. Zum einen kann man darunter den Zustand einer Anordnung von →K-Komponenten verstehen. Zum anderen wird unter Konfiguration auch der Vorgang des Konfigurierens (Konfigurationsprozess) verstanden, bei dem Komponenten parametriert und zu Anwendungssystemen zusammengesetzt werden. Um die i. Allg. riesige Menge der möglichen Kombinationsmöglichkeiten von Komponenten (→Konfigurationsraum) einzuschränken, wird dieser Vorgang vorbereitet, indem ein →Anwendungssoftwareprodukt entwickelt wird, das eine sinnvolle Teilmenge der möglichen Konfigurationen (im ersten Sinne) verkörpert.

Konfigurationsbeschreibung

Eine Konfigurationsbeschreibung ist eine in einer Konfigurationssprache (z.B. CCL) verfasste Beschreibung von sinnvollen →Konfigurationen eines →Anwendungssoftwareprodukts. Eine Konfigurationsbeschreibung beschreibt einen →Konfigurationsraum.

Konfigurationsraum

Der Konfigurationsraum ist die Menge der möglichen →Konfigurationen, die sich aus einer →Konfigurationsbeschreibung ableiten lässt.

Konfigurationszeit

Die Konfigurationszeit ist der Zeitraum innerhalb der →Definitionszeit, in dem der →Konfigurierer aus einem →Anwendungssoftwareprodukt ein Anwendungssystem konfiguriert.

Konfigurierer

Ein Konfigurierer ist eine →Rolle im →Makroprozess. Konfigurierer sind →Akteure, die →Anwendungssoftwareprodukte entsprechend den Anforderungen eines →Anwenders konfigurieren (→Konfiguration).

Kontrakt

Ein Kontrakt (Vertrag) definiert die →Signaturen (Schnittstellen) und das Verhalten einer benötigten Softwarekomponente (→S-Komponente). Eine →Schnittstelle (einer Komponente) stellt alle von außen, also von anderen Komponenten bereitzustellenden, und alle nach außen für andere Komponenten bereitgestellten Kommunikationsverbindungen dar,

über die die →Operationen der Komponente aufgerufen werden können. Zur Definition des Verhaltens einer Komponente gehört die Spezifikation der erlaubten Sequenzen (Traces) von →Kommunikationen, durch die definiert wird, wann welche →Methode/→Prozedur mit welchen Parametern aufgerufen werden darf.

Konstituierende Komponente

Siehe →Subkomponente.

Konstruktionszeit

Die Konstruktionszeit ist der Zeitraum, in dem der →Anwendungsarchitekt aus →S-Komponenten ein →Anwendungssoftwareprodukt konstruiert.

Laufzeit

Die Laufzeit (synonym: Ausprägungszeit) ist der Zeitraum, in der ein →Anwendungssystem konkret durch den Anwender genutzt wird. Die Laufzeit folgt der →Definitionszeit. Dadurch, dass moderne Architekturen es ermöglichen, zur Laufzeit eines Systems neue →Komponenten hinzuzufügen und alte wegzunehmen oder neu zu konfigurieren, überlappen sich Definitions- und Laufzeit eines Anwendungssystems zunehmend. Sie sind dann nur noch auf Ebene der Komponenten klar voneinander getrennt.

Makroprozess

An dem Entwicklungsprozess (auch: Gestaltungsprozess) komponentenbasierter →Anwendungssysteme sind mehrere →Akteure beteiligt, die in unterschiedlichen →Rollen auftreten. Der Makroprozess beschreibt die Koordination der verschiedenen am Entwicklungsprozess beteiligten Rollen (→Komponentenhersteller, →Anwendungsarchitekten, →Konfigurierer und →Anwender). Für jede dieser Rollen existiert eine Reihe von Aufgaben, die als →Mikroprozess bezeichnet wird.

Methode

Eine Methode ist eine →Operation, die innerhalb einer →Klasse gekapselt wird.

Mikroprozess

Unter einem Mikroprozess wird der Entwicklungsprozess verstanden, der durch einen →Akteur im Sinne einer spezifischen →Rolle durchgeführt wird. Der →Makroprozess ist dem Mikroprozess übergeordnet.

Operation

Eine Operation ist eine durch einen Rechner ausführbare Vorschrift zur Durchführung einer →Handlung. Es werden →Prozeduren und →Methoden unterschieden. Die Operation besteht aus einer →Signatur, die sie eindeutig kennzeichnet.

Parameterkomponente

Eine Parameterkomponente ist eine →S-Komponente, die als Parameter in den →Slot eines →Komponenten-Frameworks eingesetzt wird.

Parametrierung

Unter Parametrierung wird der Vorgang des Besetzens von Parametern (Datenfeldern) einer →S-Komponente mit Werten verstanden. Diese Parameterwerte werden zur →Definitionszeit gesetzt und schränken zur →Laufzeit die →Funktionalität der Komponente auf eine gewünschte →Funktionsweise ein. Die Parameterwerte können einfache Werte oder auch selbst wieder Komponenten sein. Die Parametrierung von mehreren Komponenten zu einem komplexen System wird auch →Konfiguration genannt.

Prädikator

Ein Prädikator ist ein Wort, dass bestimmte Gegenstände mit vergleichbarer Bedeutung bezeichnet. Gegenstände können →Geschehnisse oder (abstrakte) Dinge sein.

Prozedur

Eine Prozedur ist eine →Operation, die innerhalb einer →Komponente, aber nicht innerhalb einer →Klasse (vgl. →Methode) gekapselt wird.

R/3-Referenzmodell

Das R/3-Referenzmodell ist ein Katalog der Informationsstrukturen des R/3-Systems sowie typischer →Geschäftsprozesse, die mit dem R/3-System abgewickelt werden können. Die Geschäftsprozesse werden graphisch als →Ereignisgesteuerte Prozessketten dargestellt.

Rolle

Im Rahmen des →Makroprozesses der komponentenbasierten Anwendungsentwicklung treten mehrere →Akteure in Erscheinung, die entsprechende Aufgabenblöcke (→Mikroprozesse) durchführen, nämlich die Entwicklung von atomaren oder komplexen →Komponenten, die →Konfiguration von →Anwendungssoftwareprodukten und den Einsatz von →Anwendungssystemen. Für jeden dieser Aufgabenblöcke wird eine Rolle definiert, unter der die Aufgaben zusammengefasst werden. Als Rollen werden der →Komponentenhersteller, der →Anwendungsarchitekt, der →Konfigurierer und der →Anwender definiert.

S-Komponente

Eine S-Komponente (Softwarekomponente) ist ein in sich geschlossenes, also nicht veränderbares oder erweiterbares Element der Softwareentwicklung mit vertraglich festgelegten →Schnittstellen und →Verhalten (→Kontrakt) plus explizit angegebenen Anforderungen an die Umgebung wie beispielsweise die benötigte Laufzeitumgebung

(Betriebssystem, virtuelle Maschine, Framework). Eine Komponente liegt immer in binärer Form vor. S-Komponenten können mit anderen S-Komponenten parametriert (→Parametrierung) werden, wodurch neue, zusammengesetzte S-Komponenten entstehen. Parametrierbare Komponenten, deren Parameterwerte andere Komponenten sind, werden auch als →Komponenten-Frameworks bezeichnet. Zusammengesetzte Komponenten werden auch als komplexe Komponenten bezeichnet.

Schnittstelle

Eine Schnittstelle einer →S-Komponente ist die Menge der →Signaturen der Komponente, über die die →Operationen einer Komponenten aufgerufen werden können. Die Schnittstelle gliedert sich in die direkte und die indirekte Schnittstelle. Die direkte Schnittstelle ist die Teilmenge an Signaturen einer Komponente, deren Operationen direkt aufgerufen werden. Diese Operationen heißen →Prozeduren. Hingegen ist die indirekte Schnittstelle die Teilmenge der Signaturen, deren Operationen nur indirekt aufgerufen werden können, also nachdem zuvor die direkte Schnittstelle genutzt wurde. Diese Operationen heißen dann auch Methoden. Die Prozeduren sind vergleichbar mit statischen Methoden in objektorientierten Sprachen.

Signatur

Die Signatur bezeichnet eine →Methode oder eine →Prozedur innerhalb einer →Klasse, einem Modul oder einer →Komponente eindeutig. Sie setzt sich aus ihrem Namen sowie den Typen und den Positionen der einzelnen Argumente der Methode/Prozedur zusammen.

Slot

Siehe →Komponenten-Framework.

Softwarekomponente

Siehe →S-Komponente.

Subkomponente

Eine Subkomponente (= →konstituierende Komponente) ist eine →Komponente, die zum Aufbau einer komplexen Komponente eingesetzt wird und dann Teil dieser Komponente ist.

Symbol

Ein Symbol (Bezeichner) ist ein Zeichen oder eine Folge von Zeichen (Zeichenkette), die stellvertretend für irgend ein Ding oder einen Sachverhalt stehen kann. Symbole können visuell oder akustisch vermittelt werden.

Verhalten, aufrufbares

Das aufrufbare Verhalten einer →Komponente ist das Verhalten, was eine Komponente mindestens zur Verfügung stellen muss. Stellt sie z.B. eine Operation nicht zur Verfügung, die an einer →Schnittstelle zu einem bestimmten Augenblick benötigt wird, so ist ihr Verhalten nicht kompatibel zu dem erwarteten aufrufbaren Verhalten.

Verhalten, beobachtbares

Im Gegensatz zum →aufrufbaren stellt das beobachtbare Verhalten das Verhalten dar, das eine →Komponente maximal bieten darf. Wenn Sie mehr „Verhalten bietet“ – z.B. einen Operationsaufruf zu einem bestimmten Zeitpunkt erlaubt, der in der Umgebung dieser Komponente gar nicht erlaubt sein sollte –, so stellt diese Komponente u.U. eine Sicherheitslücke dar.

Zustand

Siehe →Ereignis.

Index

A

Ablauforganisation 94
ABox 247
Abstract-Factory-Muster 54
Abstraktion
  Black-Box-~ 45, 112
  Grey-Box-~ 45
  White-Box-~ 45
Adjunktor 101
Administrationssystem 23
Akteur 251
Aktor 37
Aktualisierung 95
Ambiguität 70
Anforderung 158
Annotation 144
  ~ von Ausdrücken 149
  ~ von Zuständen 153
  Symbol~ 149
Anwender 9, 109, 251
Anwendung 6, 8
  ~sarchitekt 9, 109
  ~sebene 197
  ~skomponente 5, 24
  ~slogik 5
  ~ssoftwareprodukt 6, 13, 19, 156, 251
  ~ssystem 6, 13, 119, 251
  ~ssystem, betriebliches 18
  ~ssystem, komponentenbasiertes 33
  ~swissen 158
Apprädikator 97
Arbeitsposten 186
ARIS 71, 251
Attributausdruck 160
Attributierung 160
Aufbauorganisation 19, 94
Aufforderung 96, 144
Ausdruck 96
Ausdrucksstärke 65
  formale ~ 65
Aussage 144
  ~form 144

Ä

Äußerungstyp 144

A

Auswahl 126
Auswahlregeln 56

B

Baustein 33
Behauptung 144
Behaviour
  invocable ~ 63, 114, 260
  observable ~ 63, 260
Behavioural View 83
Benutzungsebene 197
Beschreibungslogik 160, 245, 246
Bewegung 94
  ~saussage 97
Bewegungsdaten 5
Black-Box-Abstraktion ..... *Siehe* Abstraktion
Business Application Programming Interfaces 42

C

Callback
  ~-Beziehung 52
  ~-Verfahren 39
Checkliste 30
Checkout/Checkin-Modell 55
Common Object Request Broker Architecture
  (CORBA) 36
Component Configuration Language (CCL) 156
Component Software 40
Concept 246
Constraint ..... *Siehe* Konsistenzbedingung
Constraint Satisfaction 161
Customizing
  ~-Daten 25
  ~-Daten 5
  ~-Transaktion 25

D

Darwin 83
Datensicht 71
Datentyp 35
Definition 103
Definitionszeit 24, 252
Deklarationszeit 34, 252
Deployment Descriptor 48
Description Logic ..... *Siehe* Beschreibungslogik

Dingapprädikator 97
Disjunktor 101
Diskurs 246
Dispositionssystem 23
Distributed Common Object Model (DCOM) 36
DL-System 161, 167
Drei-Ebenen-Architektur 36

E

ECA-Regel 79, 127, 242
Effektivität 65
Eigenprädikator 96
Eigenschaft 246
Elementarsatz 97
empraktisch 92
Enterprise JavaBeans 40
Entitätstyp
  EPK-~ ..... *Siehe* ARIS
Entwicklung 6
Entwurfsmuster 39, 252
epipraktisch 92
Ereignis 81, 95, 252
  EPK-~ ..... *Siehe* ARIS
Ereignisgesteuerte Prozesskette 71, 252
Erweiterung
  orthogonale ~ 41
  parallele ~ 41
  rekursive ~ 41

F

Fachkomponente ..... *Siehe* Komponente
Fachübersetzer 199
Feature-Logik 160
Frage 144
Framework 39, 252
  Klassen-~ 39, 254
  System-~ 7
Funktion
  EPK-~ ..... *Siehe* ARIS
Funktionalität 42, 253
Funktionsbibliothek 39
Funktionssicht 71
Funktionsweise 42, 253

G

Gegenstandseinteilung 94
Geschäftsprozess 20, 253
  ~modell 20, 253
  ~modellierung 251, 253
  Soll-~ 180
  Sprachen zur ~modellierung 71
Geschehnis 94, 254
  ~apprädikator 97
  ~aussage 97
Glueing 46, 83
Guarded Choice 240

H

Handlung 94, 254
  Herstellungs~ 96
  poietische ~ 96
Hot Spot 39, 254

I

Imperativsatz 97
Import 82
Independent Software Vendor (ISV) 40
Indikativsatz 97
Individualsoftware 21
Individuum 167
Information Hiding 34
Informationssystem 18
  betriebliches ~ 254
Informationsverarbeitung
  integrierte ~ 22
Integration
  Daten~ 22
  Funktions~ 22
Interaktion 37, 62, 95, 254
  SCDL-~ 140
Interpretation 65, 143, 148

K

Kapselung 34
K-Komponente 254
  komplexe ~ 162
Klasse 254
  ~nbibliothek 4, 39
  ~ndiagramm 77
  ~n-Framework 4
Kohäsion 44
Kommunikation 37, 95, 255
Kompatibilitätsinformation 157
Komponente 24, 112, 255
  ~n-Framework 5, 7, 40, 54, 119, 122, 255
  ~nhersteller 9, 109
  ~ninstanz 47, 157, 255
  ~nmodell 109, 118

~ntypen ... 157
~nwissen ... 172
Basis~ ... 6
Definition ... 38
geschlossene ~ ... 7, 119
komplexe ~ ... 119, 255
konstituierende ~ ... 257, 259
Markt für ~n ... 54
Parameter~ ... 258
Komposition ... 255
Kompositionalität ... 70, 109
Kompositum ... 128, 256
Konfiguration ... 6, 8, 25, 157, 158, 256
~ von Softwarekomponenten ... 159
~ von Standardsoftware ... 24
~salternativen ... 110
~saufgabe ... 157
~sbeschreibung ... 256
~shilfsmittel ... 28
~sinformationen ... 112
~skomponente ... 161
~splan ... 158
~sprozess ... 157
~sraum ... 10, 169, 256
~szeit ... 24, 43, 256
abstrakte ~ ... 158
gebundene ~ ... 158
generische ~ ... 158
geschäftsprozessorientierte ~ ... 13, 179
Konfigurator ... 199
Konfigurierer ... 9, 109, 256
Kongruenzrelation ... 237
Konjunktor ... 101
Konnektoren
EPK-~ ... *Siehe* ARIS
Konsistenzbedingung ... 56, 57
Konstruktionszeit ... 257
Kontrakt ... 39, 112, 256
Kontrollsystem ... 23
Kontrollwissen ... 158
Kopplung ... 44
Kopula ... 98

L

Laufzeit ... 25, 257
Lexikon ... 92, 155
Logik erster Stufe ... 160
Lokalität ... 44

M

Makroprozess ... 6, 257
Managements by Parameters ... 28
Mehrfachvererbung ... 79
Mereologie ... 79
Methode ... 34, 69, 257
Mikroprozess ... 6, 257
Modell ... 76
Modul ... 24, 43

N

Naming-Service ... 53
Negator ... 101
NEOCLASSIC ... 161, 173
Nominator ... 96
Normsprache ... 10, 89, 92
Grammatik der ~ ... 101

O

Object Factory ... 42
Object Management Group (OMG) ... 36
Objekt ... 37
~identität ... 34
~typ ... 120
Objektorientierte Softwareentwicklung
Sprachen zur ~ ... 77
Open Application Group (OAG) ... 41
Operation ... 257
Organisationseinheit
EPK-~ ... *Siehe* ARIS
Organisationssicht ... 71
Orthogonalitätsprinzip ... 65
Orthosprache ... 10

P

Package ... 43
Paket ... 78, 82
Paketdiagramm ... 78
Parameter ... 24
parametrieren ... 7
Parametrierung ... 8, 21, 25, 258
Partikel ... 97
Persistenzebene ... 197
Pict ... 231
Core ~ ... 233
PI-Kalkül
asynchroner ~ ... 233
synchroner ~ ... 231

Planungssystem 23
Polymorphie 34
Prädikator 96, 258
Pre-/Post-Conditions 49
pre-customizing ..... *Siehe* Vorkonfiguration
Produktfamilie 39
Prozedur 69, 258
Prozess
~agent 232
~partikel 70
~sicht 71
Betriebssystem~ 37
Prozesswegweiser
EPK-~ ..... *Siehe* ARIS
Publish/Subscribe-Verfahren 51
Pull-Modell 50
Push-Modell 50

R

R/3-Referenzmodell 74, 258
Reduktionsrelation 237
Re-entrance 50
Referenzmodell
Informations~ 76
Organisations~ 76
Software~ 76
Vorgehens~ 76
Regung 94
Relator 100
Release 58
~-Fähigkeit 28
Repository 202
Request/Reply-Beziehung 52, 144
Return
~-Kanal 130, 233, 237
~-Wert 51
Revision 54
Revision Control System 55, 218
Role 246
Rolle 258

S

San Francisco 157
Satzart
Komplexe ~en von Normsprachen 101
Schema 95
Schnittstelle 259
~nvererbung 35
Direkte ~ 48
Indirekte ~ 48
Schreibung 96
SCM-Modell 159
Scripting 46
Seinsaussage 97
Semantik
~ von Zustandsdiagrammen 81
Service View 83
Signatur 259
Simple Component Description Language (SCDL) 118
Simulator 198
S-Komponente 161, 258, 259
Slot 7, 119, 259
Softwarearchitektur
Sprachen zur Beschreibung von ~en 83
Softwarekomponente ..... *Siehe* S-Komponente
Softwarekonfigurationsmanagement 54
Softwarereferenzmodell 29
Spezialisierung 110
Stammdaten 5
Standardsoftware 21
Struktur 36, 94
Sub(daten)typen 35
Subjunktor 101, 104
Substituierbarkeit 62
Substitution 147, 234
Subsumtion 250
Subtyping
Behavioural ~ 35, 63
Structural ~ 35
Symbol 259
Symbolannotation ..... *Siehe* Annotation
Systemmodell 55

T

Tataussage 97
TBox 246
Terminologie 103
terminologische Logik ..... *Siehe* Beschreibungslogik
Terminus 103
Thread 37
Tiefenstruktur 101
Trace 62, 180, 187
formaler 139
SCDL-~ 140
Trägerhandlungsschema 96
Transaktion 24

betriebliche ~ 29, 186
Transition 79
Typ 95
Typing ..... *Siehe* typisieren
typisieren 34

U

UND/ODER-Graph 56, 158
Unified Modeling Language (UML) 77
Unifikator 147
allgemeinster ~ 147
Unternehmensarchitektur 21
Unternehmensplan 20
Unterspezifikation 115
Update 28
Upgrade 28

V

Variante 54
Vererbung 34
Implementierungs- und Schnittstellen~ 112
Verhalten 36, 94
aufrufbares ~ 260
beobachtbares ~ 260
Verhaltensbeschreibung 60
Kategorien von ~en 61
Verknüpfungsoperatoren
EPK-~ ..... *Siehe* ARIS
Version 54
Versprechen 104
Verständlichkeit 111
Vorgang 95
Vorgehensweise
Bottom-up-~ 20
Top-down-~ 20
Vorkonfiguration 9, 31
branchentypische ~ 9

W

White-Box-Abstraktion ..... *Siehe* Abstraktion
Wiederverwendung 4
Wiring Standards 41
Work-Item 186
Wortart 96

Z

Zustand 79, 260
~saussage 97
~sdiagramm 77

L E B E N S L A U F

Persönliche Daten

Name	Jörg Ritter
Geburtsdatum	09. Januar 1965 in Oldenburg
Staatsangehörigkeit	deutsch
Familienstand	verheiratet
Wohnort	Harmsweg 16, 26125 Oldenburg
Telefon	0441 74351
E-Mail	joerg.ritter@o-s-c.de

Schulische Ausbildung

1971 - 1975	Grundschule Etzhorn, Oldenburg
1975 - 1977	Orientierungsstufe am Flötenteich, Oldenburg
1977 - 1984	Altes Gymnasium Oldenburg Abschluss: Allgemeine Hochschulreife

Wehrdienst

1984 - 1985	Luftwaffe, Goslar und Delmenhorst

Studium

1985 - 1991	Studium der Informatik (Nebenfach Betriebswirtschaftslehre) an der Carl von Ossietzky Universität Oldenburg; Diplomarbeit zum Thema "Entwicklung einer Lernarchitektur für den Wissenserwerb und die Wissensoptimierung" bei Prof. Dr. C. Möbus Abschluss: Diplom-Informatiker

Berufstätigkeit

1987 - 1991	Wissenschaftliche Hilfskraft bei Prof. Dr. C. Möbus an der Carl von Ossietzky Universität Oldenburg
1992 - 1993	Software-Entwickler der IDS Prof. Scheer GmbH, Saarbrücken
1993 - 1994	Bereichsleiter der IDS Prof. Scheer GmbH, Saarbrücken
1995 - 1999	Wissenschaftlicher Mitarbeiter im FuE-Bereich „Geschäftsprozess-modellierung und Referenzmodelle“ am Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -systeme (OFFIS) bei Prof. Dr. H.-J. Appelrath
seit Januar 2000	Leiter des Bereichs „Standard Systems“ der OSC - OFFIS Systems and Consulting GmbH, Oldenburg
Mai 2000	Promotion zum Doktor der Naturwissenschaften am Fachbereich Informatik der Carl von Ossietzky Universität Oldenburg

Oldenburg, den 11. September 2000