Zur Homepage der Dissertation

Semantische Komponentensuche auf Basis von Geschäftsprozessmodellen

Dissertation
zur Erlangung des Grades des
Doktors der Naturwissenschaften
am Department für Informatik
der Carl von Ossietzky Universität Oldenburg

vorgelegt von
Dipl.–Inform. Thorsten Teschke

Erstgutachter: Prof. Dr. H.-J. Appelrath
Zweitgutachter: Prof. Dr. W. Hasselbring

Datum der Disputation: 11. November 2003

Danksagung

Die Anfertigung einer Dissertation von den ersten Ideen bis zur schriftlichen Ausarbeitung nimmt im Allgemeinen einen langen Zeitraum in Anspruch. In dieser Zeit tragen eine Vielzahl von Menschen mit inhaltlichen Diskussionen und kritischen Hinweisen, aber auch Aufmunterung und Ausgleich zum Gelingen bei. Da eine Aufzählung dieser Menschen fast zwangsläufig unvollständig sein muss, beschränke ich mich auf die Nennung derer, die in meiner Erinnerung die wohl größten Beiträge zum erfolgreichen Abschluss dieser Arbeit geleistet haben.

Ein erster Dank geht an Herrn Prof. Appelrath für seine umsichtige Betreuung und Förderung. Er hat mir in seiner Rolle als Doktorvater hilfreiche Leitlinien aufgezeigt, mir aber auch die nötigen wissenschaftlichen Freiräume gewährt. Herrn Prof. Hasselbring danke ich für die Übernahme der Zweitbegutachtung.

Da diese Dissertation im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am OFFIS (Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme) entstanden ist, möchte ich mich ausdrücklich bei meinen Bereichsleitern Rolf Beyer und Christoph Mayer für die Freiräume bedanken, die sie mir eingeräumt haben.

Für zahlreiche Diskussionen danke ich Jörg Ritter, der meine Arbeit durch seine Dissertation gerade zu Beginn maßgeblich beeinflusst hat, sowie meinem Kollegen Holger Jaekel, der darüber hinaus auch zur Realisierung meines Ansatzes im KOSOBAR-Projekt beigetragen hat. Neben Holger Jaekel sind in diesem Zusammenhang Martin Keilers, der in seiner Diplomarbeit erste Ideen weiterentwickelt hat, sowie die wissenschaftlichen Hilfskräfte Stefan Brüggemann, Daniel Süpke und Hannes Winkelmann dankend zu erwähnen. Weniger spannend, aber nichtsdestotrotz wichtig ist das Korrekturlesen einer Arbeit. Hier geht mein ausdrücklicher Dank an Holger Jaekel, Thomas Aden und Arne Harren.

Neben allen inhaltlichen Beiträgen ist schließlich auch das private Umfeld von nicht zu unterschätzender Bedeutung für das Gelingen einer Dissertation. Ich möchte mich daher ganz herzlich bei meinen Eltern, meiner Schwester und Astrid Teiwes für ihre Unterstützung und ihr Verständnis bedanken.

Thorsten Teschke — Oldenburg, im Dezember 2003

Zusammenfassung

Seit dem Aufkommen von Komponententechnologien wie COM+ und Enterprise JavaBeans (EJB) gegen Ende der 90er Jahre setzt sich das Paradigma der komponentenbasierten Softwareentwicklung in zunehmendem Maße durch. Der Entwicklung von Komponentensoftware liegt die Idealvorstellung zugrunde, die Vorteile von Individual- und Standardsoftware miteinander zu kombinieren, indem Software durch die kundenindividuelle Komposition und Anpassung standardisierter Komponenten „konstruiert“ wird. Die Suche nach wiederverwendbaren Komponenten, die auf Komponentenmärkten angeboten werden, gehört dabei zu den zentralen Merkmalen komponentenbasierter Softwareentwicklungsprozesse.

Angesichts zunehmenden Wettbewerbsdrucks lässt sich seit den 90er Jahren eine Hinwendung vieler Unternehmen zu einer verstärkt prozessorientierten Ausrichtung ihrer Organisationsstrukturen und der sie unterstützenden Informationssysteme feststellen. Da fachliche Anforderungen, die infolgedessen verbreitet in Form von Geschäftsprozessmodellen festgehalten werden, bislang weder in der Praxis der Komponentensuche noch in der Forschung auf diesem Gebiet adäquate Berücksichtigung finden, ergibt sich vielfach die Notwendigkeit der Anpassung von Geschäftsprozessen an die gefundenen Komponenten.

Die vorliegende Arbeit hat sich daher zum Ziel gesetzt, die Präzision der Komponentensuche durch die explizite Berücksichtigung der in Geschäftsprozessmodellen festgehaltenen fachlichen Anforderungen zu erhöhen. Der Kern des in dieser Arbeit vorgestellten Ansatzes besteht in der Übertragung des Prinzips des Behavioural Subtyping auf die Komponentensuche. Über die Bestimmung von Subtyp-Beziehungen zwischen Geschäftsprozessmodellen und Komponentenbeschreibungen werden dabei Komponenten identifiziert, die die Ausführung der spezifizierten Geschäftsprozesse durch geeignete Funktionalität unterstützen können. Zur Beschreibung von Komponenten wird mit CDL (Component Description Language) eine Sprache vorgestellt, die eine fachlich orientierte, integrierte Darstellung der Struktur und des Verhaltens von Komponenten ermöglicht. Die eigentliche Komponentensuche beruht dann auf dem Vergleich von Geschäftsprozessmodellen mit CDL-Komponentenbeschreibungen auf der Protokoll- und der Semantikebene. Die in dieser Arbeit entwickelten Konzepte werden in Form eines Internet-basierten Komponentenbrokers prototypisch umgesetzt und anhand einer Fallstudie aus dem Bereich der kommunalen Verwaltung evaluiert.

Abstract

Since the introduction of component technologies like COM+ and Enterprise JavaBeans (EJB) in the late 90ies the paradigm of component-based software development has increasingly gained importance. The development of component software aims at combining the advantages of individual and standard software by „constructing“ software through customer-specific composition and adaptation of standardized components. Among the central characteristics of component-based software development is the retrieval of reusable components which are offered on component markets.

Due to increasing competition, in the 90ies enterprises have adopted a process-oriented perspective on their organization and the supporting information systems. Business requirements captured in business process models, however, have so far not been regarded adequately in the field of component retrieval, neither in practice nor in research. Consequently, business processes often need to be adapted to the retrieved components.

This thesis therefore aims at enhancing the precision of component retrieval by explicitly regarding the requirements captured in business process models. The transfer of the principle of behavioural subtyping to component retrieval represents the core of the approach proposed in this thesis. Components that are capable of supporting the specified business processes are discovered by analyzing subtype relationships between business process models and component descriptions. The description of components is accomplished using CDL (Component Description Language), a language facilitating domain-oriented, integrated representations of component structure and behaviour. Component retrieval is then based on the comparison of business process models and CDL component descriptions on the protocol and semantic level. The concepts developed within this thesis are prototypically realized in an internet-based component broker and evaluated using a case study from the domain of communal administration.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Komponentenbasierte Softwareentwicklung

Als gemeinsames Merkmal „reifer“ Ingenieurdisziplinen wie z. B. der Elektrotechnik oder dem Bauwesen wird vielfach die Verwendung eines Komponentenbegriffs angesehen [Gri98, HS00, Szy02]. Entwicklungs- und Herstellungsprozesse werden dabei weitgehend auf die konstruktive Kombination bestehender Komponenten zurückgeführt, die in standardisierter Form und großer Variantenvielfalt (z. B. hinsichtlich Maßen und Qualitätsstufen) von Betrieben der Zulieferindustrie, den „Komponentenherstellern“, auf einem Markt für (Ersatz-)Teile angeboten werden. Die Einführung vergleichbarer Konzepte in die Softwareentwicklung werden bereits seit der so genannten „Softwarekrise“ Ende der 60er Jahre gefordert. Zu den ersten und bekanntesten Aufsätzen zu diesem Thema gehört der Beitrag von McIlroy [McI69] zur Software-Engineering-Konferenz der NATO aus dem Jahre 1968, in dem er die Einführung eines Komponentenbegriffs in die Softwareentwicklung vorschlug: Spezialisierte Entwickler sollten Familien verwandter Softwarekomponenten anbieten, die durch Parametrisierung an spezifische Anforderungen anpassbar sind und somit in einer Vielzahl verschiedener Entwicklungsprojekte eingesetzt werden können.

Seitdem war der Begriff der Softwarekomponente als Abstraktionseinheit der Softwareentwicklung stetigem Wandel unterworfen [Cob01]. Verstand McIlroy unter einer Softwarekomponente noch eine einzelne Routine, so wurden mit den Programmiersprachen Simula 67 und Modula-2 Klassen bzw. abstrakte Datentypen als Abstraktionseinheiten in die Softwareentwicklung eingeführt. Zwar konnten sich weder Simula 67 noch Modula-2 im industriellen Einsatz durchsetzen, die in Simula 67 verwirklichten Konzepte beeinflussten jedoch maßgeblich die Entwicklung der Programmiersprache Smalltalk-80, die zu Beginn der 80er Jahre vorgestellt wurde und dem Paradigma der Objektorientierung schließlich gegen Ende der 80er Jahre zum Durchbruch verhalf: Objekte galten fortan als Bausteinkonzept der Softwareentwicklung [Bal00]. Die wachsende Komplexität verteilter Objektsysteme führte schließlich seit Mitte der 90er Jahre zur Entwicklung der heutigen Komponententechnologien, die Abstraktionseinheiten gröberer Granularität durch Entwicklungs- und Ausführungsplattformen (Application Server) unterstützen. Aktuell werden verbreitet Web Services [STK02] als neuer Erscheinungstyp von Softwarekomponenten diskutiert.

Obwohl Komponententechnologien wie COM+ und Enterprise JavaBeans (EJB) nun bereits seit mehreren Jahren produktiv in der Softwareentwicklung eingesetzt werden, existiert bislang noch keine einheitliche Definition des Begriffs der Softwarekomponente (vgl. z. B. [ND95, OHE96, Sam97, DW98, HS00, ABC+02, OMG03]). Die wohl größte Anerkennung hat die 1996 im Rahmen eines ECOOP-Workshops formulierte Definition erfahren, an deren überarbeitete Fassung [Szy02] wir uns mit unserem Komponentenbegriff im Rahmen dieser Arbeit anlehnen. Unter einer *Softwarekomponente* (im Folgenden nur noch als Komponente bezeichnet) verstehen wir eine Abstraktionseinheit relativ grober Granularität, die eine in sich geschlossene, vermarktbare Lösung für einen abgegrenzten Problembereich anbietet. Komponenten werden als ausführbare Einheiten zur Verfügung gestellt, die aufgrund expliziter Spezifikationen von Schnittstellen und Kontextabhängigkeiten im Sinne einer *Black-Box-Wiederverwendung*, d. h. ohne Zugriff auf Implementierungsdetails, durch Dritte genutzt werden können.

Der Begriff der Komponente lässt sich weiter in GUI-, System- und Fachkomponenten differenzieren. *GUI-Komponenten* sind visuelle Komponenten, die der Konstruktion graphischer Benutzungsoberflächen (Graphical User Interface, GUI) dienen und folglich auf Clientseite eingesetzt werden. Im Gegensatz zu *Systemkomponenten*, die „horizontale“, also vom betrachteten Anwendungsbereich unabhängige Dienste anbieten, sind *Fachkomponenten* „vertikale“ Komponenten, die eine fachliche, z. B. betriebswirtschaftliche Anwendungslogik implementieren. Abhängig vom primären Einsatzgebiet einer Fachkomponente lassen sich des Weiteren *Geschäftsobjektkomponenten* für den Zugriff auf persistent gehaltene Daten (*Geschäftsobjekte*) wie z. B. Aufträge oder Kunden und *Prozesskomponenten* zur Unterstützung der Abwicklung von Geschäftsprozessen wie z. B. der Auftragsbearbeitung unterscheiden. Sowohl System- als auch Fachkomponenten können client- und serverseitig eingesetzt werden. Im Rahmen dieser Arbeit konzentrieren wir uns jedoch auf serverseitige Fachkomponenten, wie sie in modernen betrieblichen Informationssystemen verbreitet Anwendung finden.

War McIlroys Forderung nach Komponenten in der Softwareentwicklung primär durch den Wunsch angetrieben, durch Konzepte der Wiederverwendung „das Rad nicht immer wieder neu erfinden“ zu müssen, so sind die Gründe für die Entwicklung heutiger Komponententechnologien vielschichtiger: Neben dem vordergründigen Wunsch nach Wiederverwendung haben technische Fragestellungen wie z. B. Verteilung, Persistenz und Sicherheit, aber auch Fragen des Projekt- und Risikomanagements von Softwareprojekten höhere Priorität erlangt. In der „traditionellen“ Softwareentwicklung können Individual- und Standardsoftware unterschieden werden. *Individualsoftware* bietet den Vorteil, dass sie eben individuell an die Anforderungen eines Kunden angepasst werden kann. Allerdings ist die Entwicklung von Individualsoftware „from scratch“ im Allgemeinen kostenträchtig und erfordert umfangreiche Kenntnisse in verschiedenen technischen und fachlichen Problemfeldern. Nicht selten scheitern Individualsoftwareentwicklungsprojekte aufgrund ihrer hohen Komplexität. Als *Standardsoftware* auf der anderen Seite bezeichnet man Software, die eine allgemein verwendbare Lösung (z. B. eine Auftragsverwaltung oder gar eine komplette ERP-Suite) für eine oder mehrere Branchen bzw. Betriebstypen darstellt. Die Anpassung der Standardlösung an die Anforderungen eines spezifischen Kunden wird durch das so genannte *Customizing* [AR00] erreicht. Die Verwendung von Standardsoftware trägt zur Reduzierung des Risikos auf Kundenseite bei, da Entwick-

lung und Wartung der Software vollständig bei deren Hersteller liegen. Als Nachteil von Standardsoftware ist jedoch zu nennen, dass ihre Einführung trotz umfangreicher, prinzipiell flexibler Anpassungsmöglichkeiten häufig die nicht immer gewollte Anpassung der Geschäftsprozesse eines Kunden nach sich zieht [Saw01]. Darüber hinaus gilt, dass sich durch die Verwendung von Standardsoftware im Allgemeinen keine Wettbewerbsvorteile gegenüber Konkurrenten erzielen lassen und Abhängigkeiten vom Hersteller der Standardsoftware entstehen [Kau00b, Szy02].

Der *Komponentensoftware* liegt die Idee zugrunde, die Vorteile von Individual- und Standardsoftware miteinander zu kombinieren, indem Software durch die kundenindividuelle Komposition und Anpassung standardisierter Komponenten entwickelt wird. Dabei sollen einerseits Ziele wie beschleunigte Entwicklungsprozesse, verbesserte Interoperabilität, höhere Qualität oder bessere Kontrolle von (finanziellen) Risiken durch die Möglichkeit zur Wiederverwendung standardisierter Lösungen, die auf einem Softwaremarkt angeboten werden, erreicht werden. Andererseits gestattet die Entwicklung von Software durch die kundenindividuelle Komposition von fremd- oder eigenentwickelten Komponenten dennoch die weitreichende Anpassung an die (sich ändernden) Anforderungen eines Kunden.

Bereits aus dieser kurzen Charakterisierung wird deutlich, dass Komponentensoftware andere Entwicklungsprozesse erfordert als Individual- und Standardsoftware (vgl. hierzu auch [Rit00]). Der Begriff der *komponentenbasierten Softwareentwicklung* bezeichnet die Entwicklung neuer bzw. die Anpassung bestehender Komponentensoftware durch die Komposition von Komponenten, die u. U. in vorgefertigter Form vorliegen und von unterschiedlichen Entwicklern stammen (vgl. auch [Bal00]). Abbildung 1.1 skizziert den Prozess der komponentenbasierten Softwareentwicklung. Ausgehend von einem Informationsmodell (unten links), das die für einen Anwendungsbereich relevanten Datenstrukturen spezifiziert, werden Geschäftsobjektkomponenten entwickelt. Auf Basis solcher Geschäftsobjektkomponenten bieten so genannte atomare Prozesskomponenten auf einer höheren Abstraktionsebene fachlich relevante atomare Dienste an (Modell atomarer Dienste, unten rechts). Durch Komposition von Prozesskomponenten können dabei auch komplexe Prozesskomponenten als Anbieter zusammengesetzter Dienste konstruiert werden (Modell komplexer Dienste, oben rechts). Die Kombination und der Einsatz von Prozesskomponenten innerhalb einer Anwendungsarchitektur ist auf die Unterstützung der im Organisationsmodell (oben links) festgehaltenen aufbau- und ablauforganisatorischen Anforderungen ausgerichtet und wird von diesen getrieben. Bei dieser vereinfachenden Darstellung der komponentenbasierten Softwareentwicklung sind technologische Fragestellungen wie z. B. die nach der zu verwendenden Komponententechnologie nicht betrachtet worden.

Neben Fragen, die den Entwurf einer Anwendungsarchitektur oder die verwendete Komponententechnologie berühren, ist die komponentenbasierte Softwareentwicklung insbesondere dadurch gekennzeichnet, dass organisationsinterne oder -übergreifende Softwaremärkte hinsichtlich verfügbarer (Fach-)Komponenten betrachtet werden. Sind grundsätzlich geeignete Komponenten am Markt erhältlich, muss entschieden werden, ob diese im Rahmen des Entwicklungsprojekts eingesetzt werden können und sollen. Fällt diese so genannte *Make-or-Buy-Entscheidung* zugunsten einer bereits verfügbaren Komponente aus, wird die Komponente von deren Anbieter erworben, anderenfalls ist eine Kompo-

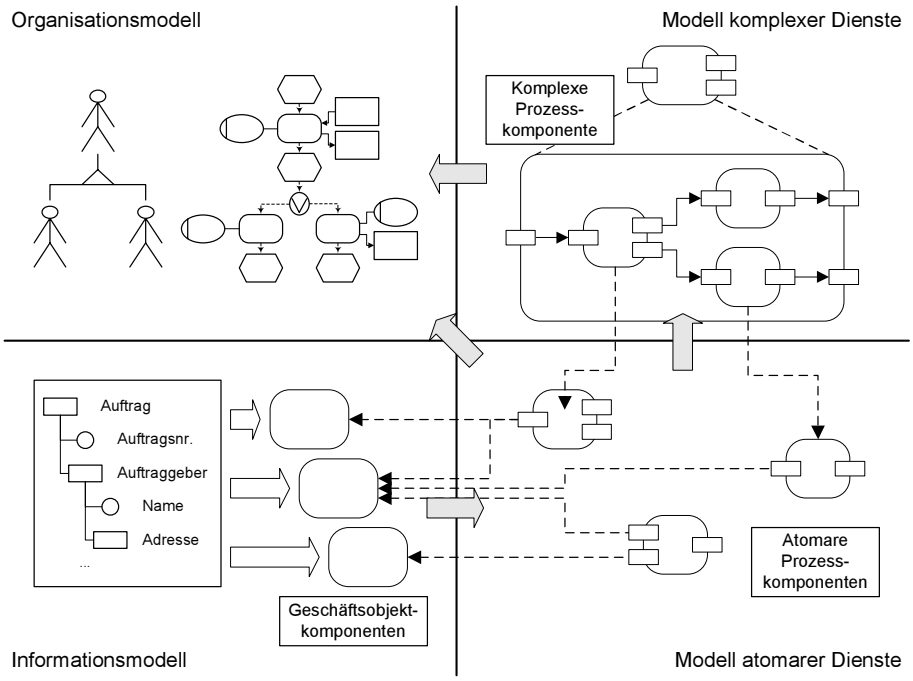


Abbildung 1.1: Komponentenbasierte Softwareentwicklung (nach [Moh03])

nente mit entsprechender Funktionalität selbst zu entwickeln. In beiden Fällen schließt sich die Integration der Komponente in das Softwaresystem und ihre kundenindividuelle Anpassung an. Das Problem der Make-or-Buy-Entscheidung wiederholt sich in der Wartungsphase durch die Änderung von Anforderungen sowie die aus der Einführung neuer Produkte und Produktversionen bzw. dem Verschwinden bekannter Produkte resultierende Dynamik an einem Softwaremarkt.

1.2 Motivation

Im vorangegangenen Abschnitt wurde die Untersuchung des Softwaremarktes im Rahmen der Make-or-Buy-Entscheidung als wichtiger Schritt komponentenbasierter Softwareentwicklung identifiziert. Seit den ersten Erfolgen beim Handel mit GUI-Komponenten (z. B. *ActiveX-Controls* oder *JavaBeans*) über das Internet sind eine Reihe elektronischer Marktplätze entstanden, über die Komponentenhersteller die von ihnen entwickelten Komponenten vertreiben können. Übersichten über solche *Komponentenmärkte* finden sich in [HM02] und [TUC03].

Biggerstaff und Richter fordern in [BR89] von einem System zur Unterstützung von Wiederverwendungsprozessen geeignete Funktionalität zum Suchen („finding“), Verstehen („understanding“), Anpassen („modifying“) und Kombinieren („composing“) wiederverwendbarer Komponenten. Ein wichtiger Erfolgsfaktor der komponentenbasierten Softwareentwicklung durch Wiederverwendung im Allgemeinen und von Komponentenmärkten im Speziellen ist somit die Fähigkeit, das Problem der gezielten Suche nach

Komponenten zu lösen. Voraussetzung für die Suche sowie die anschließende Auswahl ist dabei die Verfügbarkeit geeigneter Abstraktionen, die das Verständnis, die Bewertung und den Vergleich von Komponenten ermöglichen [Kru92].

Vergleicht man die Komponentensuche mit dem klassischen *Information Retrieval* [SM83], so lassen sich bzgl. des Suchraums sowie der bei der Suche verfolgten Zielsetzung Unterschiede feststellen. Während aktuelle Komponentenmärkte mit bis zu 10.000 angebotenen Komponenten einen durchaus noch überschaubaren Suchraum aufweisen, gilt es beim Information Retrieval aufgrund der zunehmenden Digitalisierung von Inhalten in Verbindung mit deren weltweiter Verfügbarkeit über das Internet, erheblich größere Informationsmengen effizient zugänglich zu machen.[1] Unterstützt durch Äußerungen verschiedener Autoren (vgl. [Spr00, HS00, Szy02]) gehen wir jedoch davon aus, dass sich mit zunehmender Reife der komponentenbasierten Softwareentwicklung Komponentenmärkte herausbilden werden, die schrittweise eine erheblich größere Auswahl an Komponenten anbieten. Neben dem betrachteten Suchraum unterscheiden sich Komponentensuche und klassisches Information Retrieval auch hinsichtlich der an die Suchergebnisse gestellten Qualitätsansprüche. Zwei charakteristische Größen für die Bewertung der Qualität von Ansätzen des Information Retrieval sind dabei *Präzision (Precision)*, die ein Maß für den Anteil relevanter Dokumente an den gefundenen Dokumenten liefert, und *Trefferquote (Recall)*, die den Anteil gefundener, relevanter Dokumente an allen verfügbaren relevanten Dokumenten angibt. Da mit der Erhöhung der Präzision im Allgemeinen die Trefferquote abnimmt (und umgekehrt), wird beim klassischen Information Retrieval versucht, einen möglichst guten Kompromiss zwischen diesen Kenngrößen zu erreichen. Bei der Komponentensuche ist dagegen der Präzision eine übergeordnete Bedeutung beizumessen, da man in erster Linie an der bzw. den am besten geeigneten Komponente(n) interessiert ist, um den für den Einsatz einer Komponente erforderlichen Anpassungsaufwand zu minimieren [BR89, GI94a, ZW97]. Eine Übersicht über alle Komponenten, die der Anfrage ungefähr, d. h. mit niedriger Präzision, entsprechen, ist dagegen lediglich von untergeordnetem Interesse.

Aktuelle Komponentenmärkte lösen das Problem der Komponentensuche nur in unzureichendem Maße. Die Suche nach geeigneten Komponenten wird lediglich durch relativ simple Suchverfahren wie Stichwortsuche in Freitextbeschreibungen, hierarchische Klassifikation von Komponenten oder Feldsuche unterstützt (vgl. z. B. [Com03, Sun03, Sof03, SAP03, Nom03]). Diese Suchverfahren sind zwar für Interessenten einfach zu verstehen und benutzen, berücksichtigen jedoch kaum spezifische Charakteristika von Software [MMM98]. Da ein Interessent nur sehr eingeschränkte Möglichkeiten hat, seine (fachlichen) Anforderungen an eine Komponente in den Suchvorgang einzubringen, weisen Suchergebnisse aktueller Komponentenmärkte nur eine geringe Präzision auf. Dem Interessenten fällt die Aufgabe zu, aus einer möglicherweise umfangreichen Menge gefundener Komponenten diejenige auszuwählen, die seine Anforderungen am besten erfüllt. Grundlage der Suche und anschließenden Auswahl von Komponenten bilden *Komponentenbeschreibungen*, deren Aufgabe es ist, als Abstraktionen von den angebotenen Komponenten neben Informationen zur bereitgestellten Funktionalität auch Angaben zu Hersteller, Preis, Lizenzierung, unterstützten Plattformen etc. zu machen. Da Komponentenbeschreibun-

[1] Schätzungen aus dem Jahr 1999 gehen von 800 Millionen Web-Seiten bzw. 15 Terabyte Information oder 6 Terabyte Text ohne HTML-Tags, Kommentare oder zusätzliche Leerzeichen aus [LG99].

gen in der Realität aktueller Komponentenmärkte jedoch insbesondere im Hinblick auf die angebotene Funktionalität zumeist uneinheitlich, unvollständig und missverständlich sind sowie einen informellen Charakter aufweisen, wird sowohl die automatisierte Suche nach Komponenten als auch der anschließende „manuelle“ Vergleich der Anforderungen mit den Leistungen gefundener Komponenten bzw. der direkte Vergleich gefundener Komponenten erheblich erschwert, wenn nicht gar faktisch verhindert. In Anbetracht des erwarteten Zuwachses an angebotenen Komponenten lässt sich zusammenfassend feststellen, dass die Kombination unpräziser Suchverfahren mit nur eingeschränkt aussagekräftigen Komponentenbeschreibungen die künftige Entwicklung der komponentenbasierten Softwareentwicklung im Allgemeinen und von Komponentenmärkten im Speziellen behindern wird.

Im letzten Teil dieses Abschnitts wenden wir unseren Blick von den Problemen aktueller Komponentenmärkte ab und konzentrieren uns auf zwei Beobachtungen, die bei der Komponentensuche zu berücksichtigen sind. Bei der Betrachtung der Spezifikation fachlicher Anforderungen einerseits und der Charakteristika von (Fach-)Komponenten andererseits lässt sich folgendes feststellen:

1. Seit den 90er Jahren hat sich bei der Realisierung von Softwareprojekten ein Wandel von einer bis dato primär daten- und funktionsorientierten Sicht hin zu einer verstärkt prozessorientierten Ausrichtung vollzogen. Anforderungen werden daher vielfach in Form von Geschäftsprozessmodellen [VB96] spezifiziert. Das in diesen Modellen erfasste Prozesswissen findet derzeit keine ausreichende Berücksichtigung bei der Komponentensuche.

2. Komponenten implementieren vielfach eine Anwendungslogik, die die Bereitstellung ihrer Dienste zustandsabhängig einschränkt und damit ihre Einsetzbarkeit begrenzt. Ähnlich wie bei der Einführung von Standardsoftware müssen infolgedessen die Geschäftsprozesse eines Wiederverwenders oftmals an die eingesetzten Komponenten angepasst werden.

Die Notwendigkeit der Anpassung von Geschäftsprozessen an die eingesetzte Software erscheint inakzeptabel, da gerade die Komponentensoftware den Anspruch erhebt, durch ihre Flexibilität und die Vielfalt verfügbarer Komponenten die Anpassung der Software an spezifische Anforderungen zu ermöglichen [Has02]. Eine stärkere Berücksichtigung der in Form von Geschäftsprozessmodellen spezifizierten fachlichen Anforderungen bei der Komponentensuche kann dazu beitragen, dieses Problem zu entschärfen [Saw01].

1.3 Zielsetzung und verfolgter Ansatz

Die in dieser Arbeit verfolgte Zielsetzung leitet sich direkt aus der vorangegangenen Motivation ab: Durch Berücksichtigung der in Geschäftsprozessmodellen festgehaltenen fachlichen Anforderungen soll zum einen die Präzision der Komponentensuche gesteigert und zum anderen die Bewertbarkeit gefundener Komponenten im Rahmen der anschließenden Komponentenauswahl unterstützt werden. Wir konzentrieren uns dabei auf die in Geschäftsprozessmodellen spezifizierten ablauforganisatorischen Anforderungen und lassen

evtl. vorhandene Bezüge zu Aufbauorganisation und Informationsmodell außer Acht. Dieser Ansatz zur Komponentensuche fördert die Entwicklung komponentenbasierter Software, die verstärkt auf die Ablauforganisation eines Wiederverwenders ausgerichtet ist.

Voraussetzung für die Komponentensuche auf Basis von Geschäftsprozessmodellen sind Geschäftsprozessmodelle zur Anfrageformulierung einerseits sowie umfassende Komponentenbeschreibungen andererseits, die jeweils eine klar definierte formale und fachliche Semantik aufweisen. Für die Formulierung von Geschäftsprozessmodellen greifen wir auf *lineare Prozessmodelle* [Sch01] zurück. Lineare Prozessmodelle stellen einen Ansatz zur Prozessmodellierung dar, bei dem verschiedene anerkannte Modellierungskonventionen und -richtlinien bereits im Metamodell verankert sind. Für die Beschreibung (komplexer) Komponenten nutzen wir mit der *Component Description Language (CDL)* [TR00] eine Sprache, mit der sich strukturelle und dynamische Aspekte von Komponenten integriert darstellen lassen. CDL berücksichtigt dabei, dass die durch eine Komponente implementierte Anwendungslogik die Bereitstellung ihrer Dienste zustandsabhängig einschränkt, und spezifiziert die zulässige Nutzung ihrer Dienste in Form von Interaktionsprotokollen. Dem Gedanken der *Softwarereferenzmodelle* [WWB+99, AR00] folgend betont CDL die Verständlichkeit der Beschreibungen für Fachexperten, indem die technische Sicht auf eine Komponente eng mit einer fachlichen Sicht verknüpft wird. Für die vereinheitlichte und vergleichbare Repräsentation der fachlichen Semantik von Geschäftsprozessmodellen und Komponentenbeschreibungen greifen wir auf einen normsprachlichen Ansatz [Ort97] zurück und verwenden einfache Aussagen über einer standardisierten Terminologie.

Der Kern unseres Ansatzes besteht in der Übertragung des Prinzips des *Behavioural Subtyping* [Weh02] auf die Komponentensuche. Beim Behavioural Subtyping im Kontext objektorientierter Programmiersprachen wird die Fragestellung betrachtet, ob eine gegebene Klasse hinsichtlich des durch sie definierten Verhaltens konform mit einer anderen Klasse ist. Im Kontext geschäftsprozessorientierter Komponentensuche betrachten wir eine Komponente als eine mögliche Implementierung (von Ausschnitten) eines Geschäftsprozessmodells und verwenden das Prinzip des Behavioural Subtyping, um die Konformität von Komponenten mit (Teilen von) Geschäftsprozessmodellen zu bestimmen. Dazu vergleichen wir zum einen die durch ein Geschäftsprozessmodell festgelegte Ablaufstruktur mit dem in einer Komponentenbeschreibung spezifizierten Interaktionsprotokoll, zum anderen überprüfen wir die Verträglichkeit der fachlichen Semantik einzelner Prozessschritte eines Geschäftsprozessmodells mit der fachlichen Semantik der Dienste einer Komponente.

1.4 Aufbau der Arbeit

Teil I dieser Arbeit stellt die für das Verständnis der von uns entwickelten Konzepte erforderlichen Grundlagen vor. Dazu gehört neben einer Einführung in die Grundbegriffe der Geschäftsprozessmodellierung und der Komponentensoftware auch eine Übersicht über den State of the Art auf dem Gebiet der Komponentensuche. Teil II widmet sich der detaillierten Darstellung des von uns untersuchten Ansatzes der geschäftsprozessorientierten Komponentensuche. Teil III der Arbeit stellt den Entwurf und die prototypische Implementierung unserer Konzepte sowie deren Evaluation vor, bevor Teil IV schließlich unsere Ergebnisse zusammenfasst und die Arbeit mit einem Ausblick beschließt.

Teil I

Grundlagen

Kapitel 2

Geschäftsprozessmodellierung

Ziel dieses Kapitels ist die Vorstellung der für diese Arbeit relevanten Grundlagen der Geschäftsprozessmodellierung. Im Anschluss an die Einführung der wichtigsten Grundbegriffe in Abschnitt 2.1 stellt Abschnitt 2.2 exemplarisch drei Sprachen für die Geschäftsprozessmodellierung vor, die einen unterschiedlichen softwaretechnischen Hintergrund aufweisen. Eine Zusammenfassung in Abschnitt 2.3 schließt das Kapitel mit einem Vergleich dieser Sprachen ab.

2.1 Grundbegriffe

Kürzere Innovationszyklen, gestiegene Anforderungen an die Qualität von Produkten und Dienstleistungen sowie der dadurch implizierte zunehmende Wettbewerbsdruck zwingen Unternehmen zur Steigerung der Effizienz und Anpassungsfähigkeit ihrer Aufbau- und Ablauforganisation. Seit den 90er Jahren lässt sich diesbezüglich eine Abkehr vieler Unternehmen von einer bis dato primär daten- und funktionsorientierten Betrachtung ihrer Organisationsstrukturen und der sie unterstützenden Informationssysteme und eine Hinwendung zu einer verstärkt prozessorientierten Ausrichtung feststellen. Statt der möglichst effizienten Ausführung einzelner Funktionen steht nunmehr die effiziente Ausführung oftmals organisationseinheitenübergreifender Prozesse im Mittelpunkt des Interesses [FS93, KT97]. Bei der Betrachtung solcher betrieblicher Prozesse lassen sich drei Ebenen unterscheiden [Gad02]:

- Auf der *strategischen Ebene* werden im Rahmen der Strategieentwicklung die kritischen Erfolgsfaktoren ermittelt und die Geschäftsfeldstrategie eines Unternehmens festgelegt.

- Ergebnis der (Geschäfts-)Prozessmodellierung auf der *fachlich-konzeptionellen Ebene* ist ein (Geschäfts-)Prozessmodell, das insbesondere der Dokumentation und Verbesserung betrieblicher Abläufe dient.

- Die Workflow-Modellierung auf der *operativen Ebene* führt zur Erstellung eines Workflow-Modells, das Grundlage der rechnergestützten Ausführung betrieblicher Abläufe ist.

Im Rahmen dieser Arbeit verzichten wir auf die nähere Betrachtung der strategischen Entwicklung eines Unternehmens sowie der Steuerung operativer Abläufe durch Workflows und konzentrieren uns auf die Geschäftsprozessmodellierung auf der fachlich-konzeptionellen Ebene.

Um detaillierter über Ziele und Gegenstand der Geschäftsprozessmodellierung [VB96] sprechen zu können, wollen wir zunächst die Bedeutung einiger zentraler Begriffe klären. In der Literatur ist der Terminus „Geschäftsprozess“ bereits vielfach definiert worden (für Zusammenstellungen einiger dieser Definitionen vgl. [VB96, Rum99, Gad02]). Trotz bestehender Unterschiede weisen diese Definitionen einen Konsens auf, den wir in Anlehnung an Rump [Rum99] wie folgt zusammenfassen wollen:

Definition 2.1 (Geschäftsprozess, Aktivität, Geschäftsprozessinstanz) *Ein* Geschäftsprozess *ist eine zeitlich und sachlogisch zusammenhängende, möglicherweise organisationseinheitenübergreifende Menge betrieblicher Aktivitäten. Eine* (betriebliche) Aktivität *stellt einen Schritt eines Geschäftsprozesses dar, mit dem ein bestimmtes unternehmensrelevantes Ziel verfolgt wird. Sie beinhaltet den auszuführenden Vorgang, die benutzten, manipulierten oder erzeugten Ressourcen sowie die für die Ausführung verantwortliche Organisationseinheit. Eine konkrete Ausführung eines Geschäftsprozesses bezeichnen wir als* Geschäftsprozessinstanz*.*

Beispiele für Geschäftsprozesse sind die Bearbeitung eines Kundenauftrags, die Gewährung eines Kredits oder die Abrechnung einer Dienstreise. Dabei sind Geschäftsprozesse zunächst einmal abstrakte Konzepte, die oftmals nur implizit in den Köpfen der Mitarbeiter einer Organisation existieren. Zu expliziten Beschreibungen von Geschäftsprozessen gelangt man über den Begriff des Geschäftsprozessmodells:

Definition 2.2 (Geschäftsprozessmodell, Geschäftsprozessmodellierungssprache) *Ein* Geschäftsprozessmodell *ist eine explizite Repräsentation eines Geschäftsprozesses auf der Grundlage einer Geschäftsprozessmodellierungssprache. Eine* Geschäftsprozessmodellierungssprache *ist demnach eine Sprache zur Beschreibung von Geschäftsprozessen.*

Geschäftsprozessmodellierungssprachen werden verbreitet als Diagrammsprachen angelegt, um eine höhere Verständlichkeit für Fachexperten zu erzielen.

Bzgl. der Bedeutung des Begriffs „Geschäftsprozessmodell“ gibt es unterschiedliche Standpunkte. So wird ein Geschäftsprozessmodell im Einklang mit der auf dem Gebiet der Datenbanken üblichen Terminologie vielfach als Sprache zur Beschreibung von Geschäftsprozessen betrachtet. Unser Verständnis des Begriffs wird dann durch den Begriff des Geschäftsprozessschemas wiedergegeben. Im Software Engineering werden Sprachen zur Formulierung von Modellen allgemein durch den Begriff des Metamodells beschrieben. Unter dem Begriff der Geschäftsprozessmodellierung versteht man einheitlich den Vorgang der Erstellung von Geschäftsprozessmodellen:

Definition 2.3 (Geschäftsprozessmodellierung) *Als* Geschäftsprozessmodellierung *bezeichnet man den Vorgang der Beschreibung von Geschäftsprozessen durch die Erstellung von Geschäftsprozessmodellen.*

Die von uns gewählte Definition des Begriffs des Geschäftsprozessmodells zielt auf sprachliche Konsistenz mit dem Begriff der Geschäftsprozessmodellierung ab. Als Ziele der Geschäftsprozessmodellierung nennt RUMP [Rum99]:

- *Dokumentation der Unternehmensabläufe* (Ablauforganisation), um die Zusammenhänge und Abhängigkeiten der Aktivitäten in einem Unternehmen zu explizieren.

- *Geschäftsprozessoptimierung* im Sinne einer evolutionären Verbesserung, bei der die aktuellen Prozesse im Rahmen der *Ist-Modellierung* aufgenommen, analysiert und umstrukturiert werden, oder einer revolutionären Neugestaltung [HC94], bei der die Prozesse von Grund auf überdacht und in der *Soll-Modellierung* neu modelliert werden (vgl. auch [Gad02]).

- *Simulationsstudien*, mit denen das Verhalten von Prozessen veranschaulicht oder Kennzahlen wie z. B. Durchlaufzeiten oder Kosten als Grundlage der Geschäftsprozessoptimierung ermittelt werden.

- *Strategische Informationssystemplanung* auf Basis von Geschäftsprozessmodellen durch die Bestimmung der erforderlichen IT-Unterstützung für betriebliche Abläufe, den dabei zu berücksichtigenden Informationsbedarf sowie erforderliche Schnittstellen zu weiteren Informationssystemen.

- *ISO-9000-Zertifizierung* von Herstellungs- und Entwicklungsprozessen, die im Rahmen der Geschäftsprozessmodellierung dokumentiert wurden.

- *Auswahl und Einführung von Standardsoftware* [AR00] auf der Grundlage der zu unterstützenden Geschäftsprozesse. Dabei helfen oftmals *Softwarereferenzmodelle* als Dokumentation der von einem Standardsoftwareprodukt abgedeckten Geschäftsprozesse bei der bedarfsgerechten Auswahl und anschließenden Anpassung (*Customizing*) der Standardsoftware.

- *Entwicklung von Individualsoftware* auf Basis der in Geschäftsprozessmodellen erfassten Anforderungen, sofern keine Unterstützung durch Standardsoftware möglich oder sinnvoll ist.

- *Entwicklung von Workflow-Management-Anwendungen* durch Verfeinerung und Konkretisierung der Geschäftsprozessmodelle, mit denen die Abläufe innerhalb eines Unternehmens abstrakt beschrieben werden.

Die geschäftsprozessorientierte Komponentensuche setzt auf den Einsatzmöglichkeiten der Geschäftsprozessmodellierung im Rahmen der Anforderungsermittlung und -modellierung (strategische Informationssystemplanung, Auswahl und Einführung von Standardsoftware, Entwicklung von Individualsoftware) auf. Wir verzichten daher bei der folgenden Vorstellung ausgewählter Geschäftsprozessmodellierungssprachen auf die Betrachtung ihres Einsatzes zum Zweck der Simulation und Optimierung von Geschäftsprozessen, der ISO-9000-Zertifizierung sowie des Entwurfs von Workflow-Management-Anwendungen.

2.2 Sprachen zur Geschäftsprozessmodellierung

Seit der Einführung der prozessorientierten Betrachtung betrieblicher Organisationsstrukturen und der sie unterstützenden Informationssysteme sind eine Vielzahl von Sprachen für die Geschäftsprozessmodellierung vorgeschlagen worden (vgl. z. B. die Zusammenstellungen in [VB96, Rum99]). Um ein besseres Verständnis der Ausdrucksmächtigkeit dieser Sprachen zu erhalten, wollen wir in diesem Abschnitt mit ereignisgesteuerten Prozessketten, Aktivitätsdiagrammen und linearen Prozessmodellen drei Geschäftsprozessmodellierungssprachen mit unterschiedlichem softwaretechnischem Hintergrund vorstellen. Ereignisgesteuerte Prozessketten, die insbesondere durch ihren Einsatz im R/3-Referenzmodell Verbreitung gefunden haben, basieren im Wesentlichen auf klassischen Modellierungstechniken. Aktivitätsdiagramme integrieren als Teil der Unified Modeling Language (UML) objektorientierte Ansätze in die Geschäftsprozessmodellierung, und lineare Prozessmodelle wurden primär für den Einsatz im Bereich des Workflow-Managements entwickelt.

2.2.1 Ereignisgesteuerte Prozessketten und ARIS

Scheer hat mit der *Architektur integrierter Informationssysteme (ARIS)* ein »Rahmenkonzept zur ganzheitlichen Beschreibung (Modellierung) computergestützter Informationssysteme« [Sch98b] vorgeschlagen. Dieses kurz als ARIS-Architektur bezeichnete Rahmenkonzept soll durch die Einführung verschiedener Beschreibungssichten und -ebenen helfen, die Komplexität der Modellierung betrieblicher Informationssysteme zu bewältigen (vgl. Abbildung 2.1).

Abbildung 2.1: ARIS-Architektur (nach [Sch98b])

In ihrer ursprünglichen Konzeption umfasste die ARIS-Architektur die folgenden vier Sichten (vgl. auch [Rum99]):

- Die *Datensicht* beschreibt die im betrachteten Informationssystem relevanten Informationsobjekte sowie deren Beziehungen untereinander.
- In der *Funktionssicht* werden die Vorgänge, die Input-Leistungen zu Output-Leistungen transformieren, zusammen mit ihren Abhängigkeiten modelliert.
- Die *Organisationssicht* spezifiziert die Organisationsstruktur eines Unternehmens durch die Angabe von Organisationseinheiten und deren Beziehungen.
- In der *Steuerungs-* oder *Prozesssicht* werden die Geschäftsprozesse eines Unternehmens modelliert. Dabei werden Teile der Informationen, die in den übrigen, auf statische Aspekte ausgerichteten Sichten modelliert wurden, zu einem dynamischen Modell integriert, mit dem das Verhalten des Informationssystems wiedergegeben wird.

Eine später erweiterte Fassung der ARIS-Architektur betrachtet noch eine fünfte Sicht:

- Die *Leistungssicht* enthält alle materiellen und immateriellen Input- und Output-Leistungen eines Unternehmens (z. B. Informationsdienstleistungen, Sachleistungen und Finanzmittel).

Jede dieser Sichten ist in drei Beschreibungsebenen untergliedert, auf denen das Informationssystem eines Unternehmens mit unterschiedlichem Abstraktionsgrad beschrieben wird:

- Das *Fachkonzept* beschreibt das Informationssystem aus einer stark betriebswirtschaftlich geprägten Perspektive.
- Das *DV-Konzept* überträgt das Fachkonzept in die Begriffswelt der Datenverarbeitung, ohne jedoch auf konkrete Implementierungsaspekte einzugehen.
- Die *(technische) Implementierung* stellt den Bezug zwischen dem DV-Konzept und der im Unternehmen verfügbaren IT-Infrastruktur (Hard- und Softwarekomponenten) her.

Mit Ausnahme der Leistungssicht, in der das DV-Konzept und die Implementierung mangels spezifischer Implementierungsverfahren nicht modelliert werden, stehen für jede der fünf Sichten auf den einzelnen Beschreibungsebenen Sprachen zur Verfügung, mit denen sich die jeweils relevanten Aspekte eines Informationssystems spezifizieren lassen [Sch98a]. Auf der Ebene des Fachkonzepts sind dies semantische Modelle wie z. B. das Entity-Relationship-Modell (ERM) für die Datensicht, Funktionsbäume für die Funktionssicht, Organigramme für die Organisationssicht oder Produktbäume für die Leistungssicht. In der Steuerungssicht werden *ereignisgesteuerte Prozessketten (EPK)* [KNS92] eingesetzt, die im Folgenden übersichtsartig vorgestellt werden sollen (für eine detailliertere Einführung siehe [Rum99]).

Das EPK-Modell sieht im Kern zwei Beschreibungselemente für die Modellierung von Prozessen vor. *Funktionen* beschreiben als aktive Komponenten eines Informationssystems die Durchführung betrieblicher Vorgänge, die zur Erfüllung eines Unternehmensziels beitragen. Beispiele für Funktionen sind „Rechnung prüfen“ oder „Produktionsauftrag freigeben“. *Ereignisse* stellen als passive Komponenten eines Informationssystems

das Eintreten eines betriebswirtschaftlichen Zustands eines oder einer Gruppe von Informationsobjekten innerhalb des Informationssystems dar. Sie determinieren den weiteren Ablauf in einem Informationssystem, indem sie die Ausführung von Funktionen auslösen können. Beispiele für Ereignisse sind „Produktionsauftrag freigegeben“ und „Rechnung eingegangen“. Die graphische Repräsentation von Funktionen und Ereignissen ist in Abbildung 2.2 dargestellt.

Abbildung 2.2: Grundlegende Beschreibungselemente einer EPK

Die Grundstruktur einer EPK ist durch eine alternierende Anordnung von Ereignissen und Funktionen charakterisiert.[1] Eine Funktion wird dabei einerseits immer durch ein oder mehrere Ereignisse ausgelöst, andererseits erzeugt jede Funktion ein oder mehrere Folgeereignisse. Die Modellierung komplexerer Ablauflogik wird durch *Verknüpfungsoperatoren (Konnektoren)* unterstützt, die konjunktive („AND“, $\wedge$), disjunktive („XOR“) und adjunktive („OR“, $\vee$) Verknüpfungen von Ereignissen und Funktionen gestatten (siehe Abbildung 2.3). Bei der Modellierung von Prozessen mit Hilfe dieser Verknüpfungsoperatoren ist zu berücksichtigen, dass Ereignisse keine Entscheidungsbefähigung haben und somit aufgrund eines Ereignisses nicht über einen OR- oder XOR-Konnektor zwischen anschließend auszuführenden Funktionen gewählt werden kann. Der Kontrollfluss eines mit einer EPK modellierten Prozesses ist durch die Verbindung von Funktionen, Ereignissen und Verknüpfungsoperatoren mittels gestrichelter Pfeile definiert.

Abbildung 2.3: Konnektoren

Aufgrund ihres kompakten Symbolvorrats weisen EPKs keinen Bezug zur Organisationssicht auf. Eine Beziehung zur Datensicht besteht darüber hinaus nur implizit aufgrund der Charakterisierung von Ereignissen als Zustände eines Informationssystems. Von *erweiterten ereignisgesteuerten Prozessketten (eEPK)* wird in der Literatur gesprochen, wenn die EPK-Notation um Symbole für die Repräsentation von *Organisationseinheiten* und *Informationsobjekten* ergänzt wird. Organisationseinheiten werden mittels ungerichteter Kanten mit Funktionen verbunden und spezifizieren dadurch, welche Rolle für die Ausführung der Funktion verantwortlich ist. Informationsobjekte repräsentieren Informationen, die für die Ausführung von Funktionen relevant sind. Die Richtung der Verbindung zwischen Informationsobjekt und Funktion gibt dabei an, ob das Informationsobjekt für die Ausführung einer Funktion benötigt oder aber durch deren Ausführung manipuliert bzw. erzeugt wird. Die Verknüpfung von Prozessen wird im EPK-Modell über

[1] Genau genommen müsste hier von Ereignistypen und Funktionstypen gesprochen werden, da in einer EPK keine konkreten Ereignisse oder Funktionen modelliert werden. In der Praxis und der wissenschaftlichen Literatur haben sich allerdings die hier verwendeten Bezeichnungen durchgesetzt [Rum99].

gemeinsam verwendete Ereignisse erreicht. Da diese Verknüpfungen nur implizit und daher gerade für den menschlichen Betrachter nur schwerlich ersichtlich sind, ergänzt das eEPK-Modell den Symbolvorrat außerdem um den *Prozesswegweiser*, der die Verbindung eines Prozesses mit einem Folgeprozess explizit anzeigt. Die graphische Repräsentation von Organisationseinheiten, Informationsobjekten und Prozesswegweisern ist in Abbildung 2.4 dargestellt.

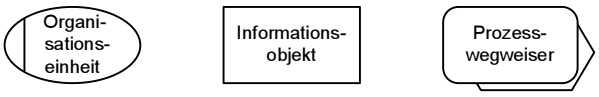


Abbildung 2.4: Ergänzende Beschreibungselemente einer eEPK

Ein weiteres Mittel zur Strukturierung von Prozessen stellt die *Hierarchisierung* dar. Sie gestattet die Verfeinerung von Funktionen, die auf relativ hohem Abstraktionsniveau beschrieben sind, indem diesen im Sinne der *hierarchischen Prozessmodellierung* eine (e)EPK mit höherem Detaillierungsgrad hinterlegt wird [KT97]. Im Folgenden verzichten wir auf die explizite Unterscheidung von EPK und eEPK und sprechen verallgemeinernd nur noch von EPKs. Als Beispiel für eine EPK zeigt Abbildung 2.5 einen Ausschnitt aus dem Geschäftsprozess „Güterbeschaffung“ des Automobilherstellers Ford.

EPKs werden als semiformale Sprachen bezeichnet, da ihre Semantik nicht formal definiert ist. Vorschläge zur Definition einer formalen Semantik für EPKs finden sich u. a. bei Keller und Teufel [KT97] sowie Rump [Rum99]. Das EPK-Modell ist aufgrund seines verbreiteten Einsatzes in der Praxis verschiedentlich erweitert worden. So hat Rosemann in [Ros96] weitere Verknüpfungsoperatoren ergänzt, die eine elegantere Modellierung komplexer Ablauflogik gestatten. In dem Ansatz der *objektorientierten ereignisgesteuerten Prozesskette (oEPK)* verbinden Scheer, Nüttgens und Zimmermann [SNZ97] das EPK-Modell mit den objektorientierten Methoden der UML. Ein Geschäftsprozess ist im oEPK-Modell als ereignisgesteuerte Bearbeitung und Interaktion von Geschäftsobjekten mit dem Ziel der Leistungserstellung definiert. Ein Geschäftsobjekt wird dabei als betriebswirtschaftliche Leistung verstanden, welche gemäß dem objektorientierten Paradigma die zur Bearbeitung relevanten Methoden und Attribute kapselt.

2.2.2 Geschäftsprozessmodellierung mit der UML

Einhergehend mit dem Durchbruch der objektorientierten Softwareentwicklung sind gegen Ende der 80er Jahre eine Vielzahl von Sprachen und Methoden für die objektorientierte Analyse und den Entwurf komplexer Softwaresysteme entstanden, von denen zu Beginn der 90er Jahre die *Object Modeling Technique (OMT)* von Rumbaugh [RBP+91], die Booch-*Methode* [Boo94] und Jacobsons *Object-Oriented Software Engineering (OOSE)* [JCJO92] die größte Akzeptanz erfahren haben. Um den damals herrschenden „Methodenkrieg“ zu beenden, hat die *Object Management Group (OMG)* als wichtigstes Standardisierungsgremium für die objektorientierte Softwareentwicklung im Jahre 1996 zur Spezifikation eines Modellierungsstandards aufgerufen [Oes01, HK03]. Aus diesem Aufruf ist die *Unified Modeling Language (UML)* hervorgegangen, die insbesondere durch die eingangs erwähnten Arbeiten von Rumbaugh, Booch und Jacobson geprägt ist, aber

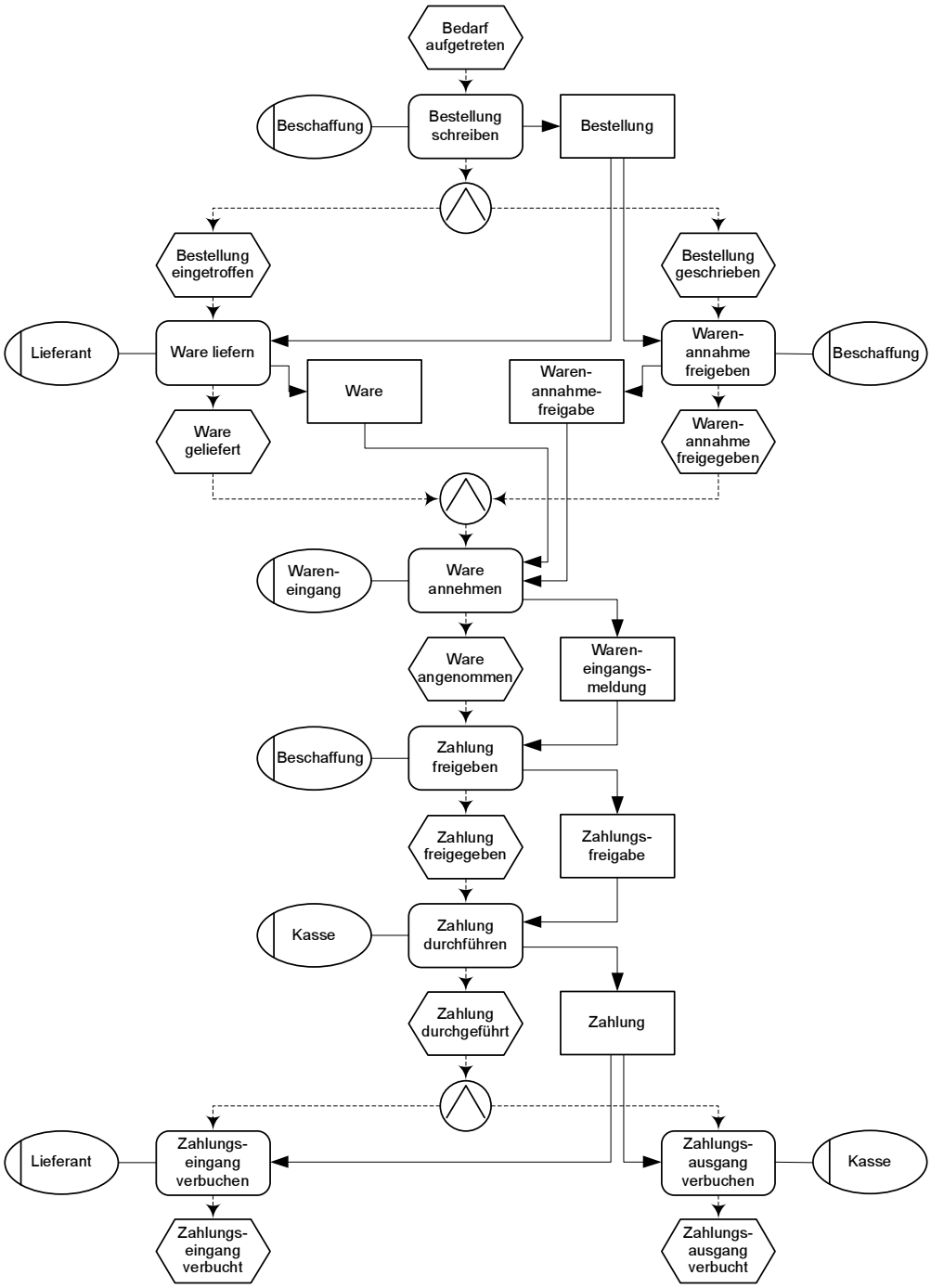


Abbildung 2.5: Beispiel einer EPK (modelliert nach [FS96a])

auch andere Techniken wie z. B. Harels *Statecharts* [HG97] integriert. Aktuell liegt die UML in der Version 1.5 [OMG03] vor.

Entgegen ihrem Namen ist die UML weniger eine einzelne Sprache als vielmehr eine Sammlung inhaltlich überlappender Sprachen, die die umfassende Spezifikation objektorientierter Softwaresysteme von der Analyse bis zu Entwurf und Implementierung unterstützen. So stehen für die Abbildung der unterschiedlichen Aspekte eines Softwaresystems acht Diagrammsprachen oder -typen zur Verfügung, mit denen sich verschiedene Sichten auf das Softwaresystem modellieren lassen (für eine Vorstellung dieser Diagrammtypen siehe z. B. [Oes01, HK03]). In den frühen Phasen der Analyse kommen dabei primär Anwendungsfalldiagramme und Aktivitätsdiagramme zum Einsatz [Oes98], deren Bezug zur Geschäftsprozessmodellierung wir im Folgenden eingehender darlegen wollen.

Ausgangspunkt der objektorientierten Softwareentwicklung mit der UML ist im Allgemeinen die Identifikation von *Anwendungsfällen*. Der Begriff des Anwendungsfalls bezeichnet dabei den Kontext und die Gliederung des Umgangs mit einem Geschäftsvorfall wie z. B. einem Beschaffungsvorgang. Anwendungsfälle beschreiben damit Anforderungen an das zu realisierende System aus der Sicht eines Systembenutzers (*was* soll das System leisten?) und abstrahieren dabei von Details des Systemverhaltens (*wie* soll das System dies leisten?) [Oes01]. Die UML unterstützt die Modellierung von Anwendungsfällen durch *Anwendungsfalldiagramme*. Diese erlauben die graphische Darstellung der Grenzen des zu modellierenden Softwaresystems, der relevanten Anwendungsfälle und ihrer Beziehungen sowie der an der Ausführung von Anwendungsfällen beteiligten Akteure. Abbildung 2.6 zeigt ein Beispiel eines Anwendungsfalldiagramms für ein System „Materialverwaltung“ mit den Anwendungsfällen „Güterbeschaffung“, „Zahlungsabwicklung“ und „Bedarfsermittlung“, an deren Ausführung die Akteure „Wareneingang“, „Kasse“ und „Beschaffung“ beteiligt sind. Nicht zuletzt, weil die graphische Notation von Anwendungsfällen mit der UML nur eingeschränkt aussagekräftig ist, sind verschiedene Schablonen für die ergänzende textuelle Beschreibung vorgeschlagen worden (vgl. z. B. [Bal00, Oes01]). Diese Schablonen umfassen u. a. Informationen zu erbrachten Leistungen, erwarteten Vorbedingungen, beteiligten Akteuren, grobem Ablauf, Sonderfällen und Fehlerbehandlung eines Anwendungsfalls.

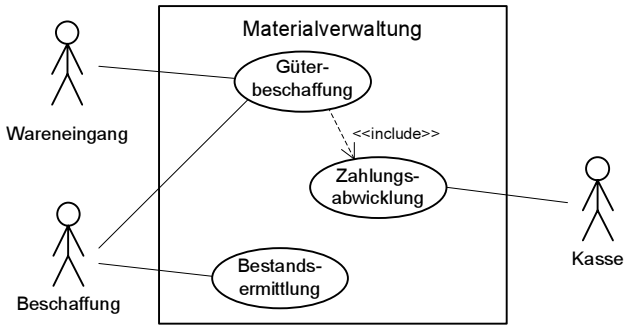


Abbildung 2.6: Beispiel eines Anwendungsfalldiagramms

Ein Geschäftsprozess kann im Kontext der objektorientierten Analyse mit der UML als Ablauf der Bearbeitung eines Geschäftsvorfalls betrachtet werden, der durch einen Anwendungsfall identifiziert und eingegrenzt wurde. Die Verfeinerung von Anwendungsfällen durch derartige Geschäftsprozesse wird in der UML durch *Aktivitätsdiagramme*

ermöglicht [Oes98]. Zentrales Element von Aktivitätsdiagrammen sind *Aktivitäten*[2], die als einzelne Prozessschritte im Laufe des modellierten (Geschäfts-)Prozesses durchzuführende (betriebliche) Vorgänge repräsentieren. Der Kontrollfluss eines Prozesses wird in Aktivitätsdiagrammen durch Transitionen spezifiziert, die als gerichtete Kanten zwischen den Aktivitäten dargestellt werden. Die Modellierung des Objektflusses wird in Aktivitätsdiagrammen durch die Möglichkeit unterstützt, die im Rahmen der Ausführung einer Aktivität benutzten, manipulierten oder erzeugten *Objekte* zu spezifizieren. Das Ein- und Ausgabeverhalten von Aktivitäten wird dabei durch gestrichelte Pfeile zwischen der Aktivität und den Objekten dargestellt. Neben der Kennzeichnung eines Objekts durch den Klassennamen und einen optionalen Objektbezeichner kann zusätzlich der Objektzustand (und damit insbesondere auch dessen Veränderung im Verlauf des Geschäftsprozesses) angegeben werden. Die organisatorische Zuordnung von Aktivitäten wird in Aktivitätsdiagrammen durch *Verantwortlichkeitsbereiche* abgebildet. Ein Verantwortlichkeitsbereich ordnet genau einem Akteur (z. B. einer Rolle oder Organisationseinheit des betrachteten Unternehmens) oder einem Objekt des zu realisierenden Systems die Verantwortung für die Durchführung einer Menge von Aktivitäten zu. Aufgrund der graphischen Darstellung als nebeneinander liegende senkrechte Bahnen werden Verantwortlichkeitsbereiche oft auch als „swimlanes“ (Schwimmbahnen) bezeichnet. Zwei weitere, eher selten verwendete syntaktische Elemente dienen der Modellierung des Empfangs und des Sendens von *Ereignissen*. Abbildung 2.7 zeigt die graphische Darstellung von Aktivitäten, Objekten, Verantwortlichkeitsbereichen und Ereignissen.

Abbildung 2.7: Grundlegende Beschreibungselemente eines Aktivitätsdiagramms

Für die Modellierung des Kontrollflusses stehen neben den bereits angesprochenen Transitionen noch weitere Sprachelemente zur Verfügung. So lassen sich alternative Entscheidungen durch die Annotation der Transitionen mit Bedingungen spezifizieren. Spezielle *Entscheidungsknoten* tragen zur Lesbarkeit eines Aktivitätsdiagramms bei, indem sie solche alternativen Verzweigungen explizit machen. Parallele Abläufe lassen sich mittels so genannter Synchronisationsbalken modellieren, mit denen der Kontrollfluss in mehrere nebenläufige Ausführungspfade aufgespalten (*Gabelung*) und später wieder zusammengeführt (*Vereinigung*) werden kann. Der Kontrollfluss eines Aktivitätsdiagramms wird wie bei Zustandsdiagrammen durch spezielle Symbole für *Start-* und *Endzustände* begrenzt. Während in einem Aktivitätsdiagramm nur ein Anfangszustand existieren darf, können aus Gründen der Übersichtlichkeit mehrere Endzustände benutzt werden. Die graphische

[2]Sofern es im Verlauf dieser Arbeit notwendig wird, explizit zwischen dem speziellen, in der UML verwendeten Aktivitätsbegriff und unserem allgemeinen Aktivitätsbegriff aus Definition 2.1 zu unterscheiden, verwenden wir für den letztgenannten Begriff die Bezeichnung „betriebliche Aktivität“.

Repräsentation der Diagrammelemente zur Modellierung des Kontrollflusses ist in Abbildung 2.8 dargestellt. Als Beispiel für ein Aktivitätsdiagramm zeigt Abbildung 2.9 analog zu Abbildung 2.5 im vorangegangenen Abschnitt einen Ausschnitt aus dem Geschäftsprozess „Güterbeschaffung“ beim Automobilhersteller Ford.

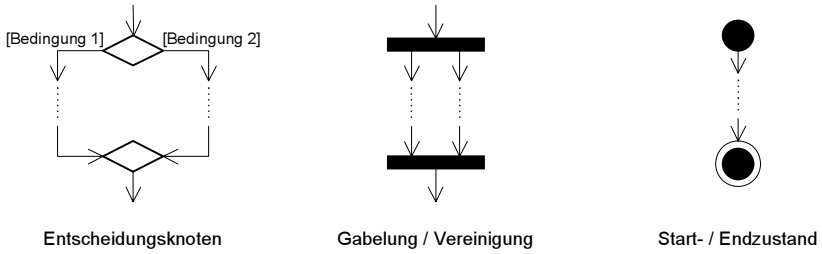


Abbildung 2.8: Elemente zur Beschreibung des Kontrollflusses in Aktivitätsdiagrammen

Die Spezifikation der UML [OMG03] erhebt den Anspruch, eine präzise und konsistente Definition der Notation und Semantik der UML zu geben. Da die Definition der Semantik (neben dem Einsatz der *Object Constraint Language (OCL)* für die Spezifikation einiger Wohlgeformtheitseigenschaften) allerdings primär auf „englischer Prosa“ basiert, lässt sie fast zwangsläufig Interpretationsspielräume offen, die Ausgangspunkt von Missverständnissen im Softwareentwicklungsprozess sein können. Die Definition einer formalen Semantik ist daher Gegenstand verschiedener Forschungsarbeiten, die teilweise in der *Precise UML Group (pUML)* [PUM03] zusammengefasst werden. Ein Vorschlag für eine präzise Definition der Semantik von Aktivitätsdiagrammen findet sich in [BCR00a].

Die UML bietet verschiedene Mechanismen an, mit denen sich die Semantik ihrer Modellelemente an spezifische Anforderungen anpassen und erweitern lässt. Ein *UML-Profil* stellt eine zusammengehörige Menge solcher Anpassungen und Erweiterungen dar, die einem gemeinsamen Zweck dienen [OMG03]. Die OMG hat mit dem *UML Profile for Enterprise Distributed Object Computing* [OMG02b] ein Modellierungsframework zur Unterstützung der Entwicklung betrieblicher Anwendungssysteme auf der Grundlage von UML 1.4 spezifiziert. Dieses Framework umfasst u. a. fünf UML-Profile, mit denen unterschiedliche Aspekte eines Softwaresystems modelliert werden können. Von besonderem Interesse aus Sicht der Geschäftsprozessmodellierung ist das *Business Process Profile*, das insbesondere in Verbindung mit dem *Entities Profile* zur Abbildung statischer Strukturen und dem *Events Profile* zur Spezifikation ereignisgetriebener Systeme für die Modellierung des Verhaltens betrieblicher Anwendungssysteme genutzt werden kann. Es bietet gegenüber den hier vorgestellten Aktivitätsdiagrammen erweiterte Möglichkeiten für die Beschreibung von Geschäftsprozessen. Dazu gehören u. a. die regelbasierte Zuordnung von Organisationseinheiten und Informationsobjekten zu Aktivitäten, die Repräsentation verschiedener temporaler Aspekte, die Spezifikation der wiederholten Ausführung von Aktivitäten und die Behandlung von Ausnahmesituationen. Darüber hinaus unterstützt das Business Process Profile die *hierarchische Prozessmodellierung* durch die Möglichkeit, komplexe betriebliche Aktivitäten einer höheren Abstraktionsebene durch Prozesse mit Aktivitäten niedriger Abstraktion zu beschreiben.

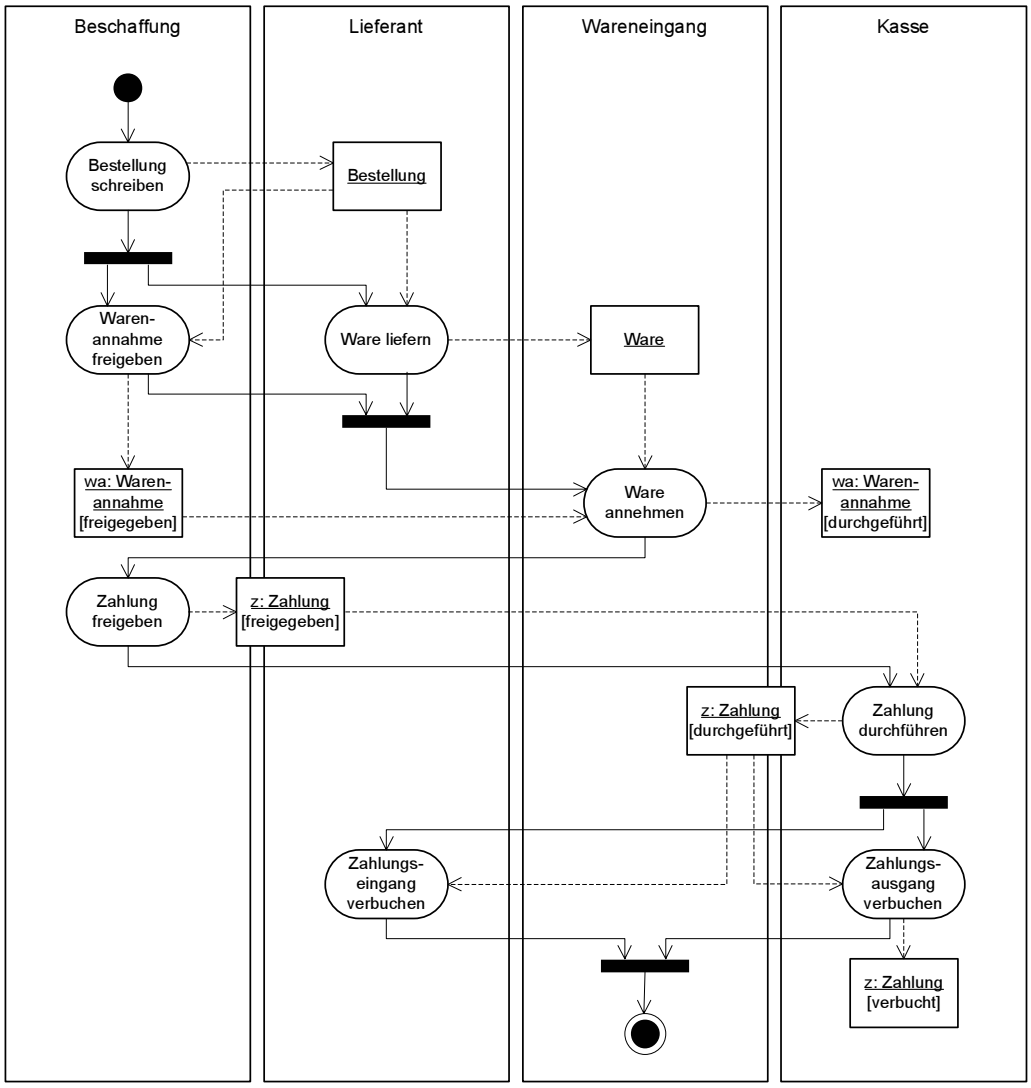


Abbildung 2.9: Beispiel eines Aktivitätsdiagramms (modelliert nach [FS96a])

2.2.3 Lineare Prozessmodelle

Ein grundsätzliches Problem der Prozessmodellierung – sei es im Bereich der Geschäftsprozesse oder der Workflows – besteht darin, dass das Prozesswissen eines Unternehmens oftmals nur implizit „in den Köpfen seiner Mitarbeiter“ vorhanden ist. Da diese Mitarbeiter als menschliche „Wissensträger“ vielfach nicht in der Lage sind, dieses organisationale Wissen mit der erforderlichen Vollständigkeit und Präzision wiederzugeben, ist die Prozessmodellierung als Form der Explikation organisationalen Prozesswissens mit erheblichem Zeitaufwand und entsprechenden Kosten verbunden [Sch01]. Das *Process Mining* [ADH+03] stellt einen Ansatz zur automatischen Rekonstruktion von Prozesswissen dar, bei dem dieses Wissen aus Aufzeichnungen über die Ausführung von

Prozessen extrahiert wird. Für die Repräsentation des extrahierten Prozesswissens setzt SCHIMM [Sch01] *lineare Prozessmodelle* ein, die auch als Grundlage der Analyse und Ausführung von Workflows dienen können. Lineare Prozessmodelle lassen sich zwischen semi-formalen Diagrammsprachen wie den bereits vorgestellten EPKs und Aktivitätsdiagrammen, formalen Diagrammsprachen wie Petri-Netzen [Rei85] oder den auf ihnen basierenden Workflow-Nets [AB02] und formalen Sprachen wie Prozessalgebren [BW90] positionieren.

Die Entwicklung der linearen Prozessmodellierung ist motiviert durch den Wunsch, neben allgemeinen Modellierungskonventionen oder -richtlinien [BRS95] wie z. B. die Verwendung von Prozessbausteinen mit definierter Semantik oder die Einhaltung linearer Kontrollstrukturen (vgl. z. B. [Rum99]) auch präzise Anforderungen an Eigenschaften von Prozessmodellen wie z. B. *Soundness* für Workflow-Nets [AB02] bereits in das Metamodell einer (Geschäfts-)Prozessmodellierungssprache einfließen zu lassen [Sch01]. In diesem Sinne stellen lineare Prozessmodelle eine kompakte Teilmenge anerkannter Modellierungspraktiken dar, die sicherstellen, dass die beschriebenen Prozesse frei von *Deadlocks (Verklemmungen)* und anderen Anomalien sind [Sch03].

Lineare Prozessmodelle werden aus Prozessbausteinen (den so genannten *Blöcken*) zusammengesetzt, die genau einen Eintritts- und Austrittspunkt besitzen. Da die resultierenden Modelle aufgrund dieser Struktur ihrer Blöcke einen linearen Kontrollfluss aufweisen, werden sie in Anlehnung an die lineare/strukturierte Programmierung [DDH72] als linear bezeichnet. Bei den Blöcken lassen sich Operanden und Operatoren unterscheiden, wobei letztere den Kontrollfluss eines Prozessmodells definieren und als Argumente weitere Blöcke erwarten. Das grundlegende Prinzip der linearen Prozessmodellierung besteht darin, durch beliebig tiefe Verschachtelung von Operatoren eine Baumstruktur aufzubauen, deren Blätter durch Operanden gebildet werden. Dabei sind mit Aktivitäten und Prozessen zwei Arten von Operanden zulässig. *Aktivitäten* sind betriebliche Vorgänge, deren Ausführung eine bestimmte Dauer hat. Diese Dauer wird im Rahmen der Aufzeichnung von Prozessinstanzen durch ein Startereignis, das den Beginn der Ausführung einer Aktivität signalisiert, und ein Endereignis, das den Abschluss ihrer Ausführung kennzeichnet, eingegrenzt. Aktivitäten können organisatorisch einer *Rolle* zugeordnet sein und für ihre Ausführung auf *Ressourcen* zugreifen. In der graphischen und prozessalgebraischen Repräsentation linearer Prozessmodelle finden Rollen und Ressourcen allerdings derzeit keine Berücksichtigung. Ihre Modellierung wird jedoch durch den *Process Miner* [Sch02], ein Werkzeug zur Extraktion und Modellierung linearer Prozesse, unterstützt. *Prozesse* oder präziser lineare Prozessmodelle können im Sinne der *hierarchischen Prozessmodellierung* als Operanden eingesetzt werden, um Prozesse auf unterschiedlichen Abstraktionsebenen zu beschreiben. Die graphische Darstellung von Aktivitäten und Prozessen ist gleichermaßen durch ein Rechteck gegeben, das mit dem Namen der referenzierten Aktivität bzw. des Prozesses beschriftet ist. In der Prozessalgebra der linearen Prozessmodellierung werden Referenzen auf Aktivitäten und Prozesse als einfache Terme repräsentiert, die den jeweiligen Bezeichner wiedergeben.

Das Metamodell der linearen Prozessmodellierung betrachtet in seiner Grundform vier Operatoren [Sch01, Sch03]. Die *sequentielle Komposition* von Blöcken legt die exakte Ausführungsreihenfolge der enthaltenen Blöcke fest. In der Prozessalgebra linearer Prozessmodelle wird die sequentielle Komposition durch den Präfixoperator $\mathcal{S}$ symbolisiert.

Die sequentielle Komposition zweier Blöcke B_1 und B_2 wird durch den Term $\mathcal{S}(B_1, B_2)$ formuliert.

Mit der *alternativen Komposition* wird die Auswahl einer von mehreren alternativen Fortsetzungsmöglichkeiten des Prozesses modelliert. Nicht gewählte Alternativen werden im weiteren Prozessablauf nicht mehr berücksichtigt. Die Auswahl einer Alternative kann an Bedingungen geknüpft werden, die dem Kompositionsoperator zugeordnet werden und ähnlich wie Rollen und Ressourcen keine graphische und prozessalgebraische Repräsentation haben. Die alternative Komposition betrachtet auch den Sonderfall, dass die Ausführung der angebotenen Alternativen gänzlich optional ist. Dazu wird der leere Prozessterm ϵ (bzw. die entsprechende Aktivität) in die Prozessalgebra eingeführt. In der Prozessalgebra wird die alternative Komposition durch den Präfixoperator $\mathcal{A}$ repräsentiert. Ihre Anwendung auf zwei Blöcke ergibt sich analog zur Anwendung der sequentiellen Komposition.

Die *parallele Komposition* verlangt ähnlich wie die sequentielle Komposition die Ausführung aller enthaltenen Blöcke. Allerdings wird dabei die Ausführungsreihenfolge der Blöcke nicht vorgegeben, d. h. die enthaltenen Blöcke können in beliebiger Reihenfolge und insbesondere auch nebenläufig ausgeführt werden. Der Kontrollfluss des Prozesses wird durch die parallele Komposition folglich aufgespalten und nach Abarbeitung der enthaltenen Blöcke wieder verschmolzen. Ebenso wie der Vereinigung paralleler Verarbeitungszweige in Aktivitätsdiagrammen liegt auch der parallelen Komposition der Gedanke einer *Und-Synchronisation* zugrunde, d. h. der dem Kompositionsoperator folgende Block kann erst ausgeführt werden, wenn die Bearbeitung *aller* in der parallelen Komposition enthaltenen Blöcke abgeschlossen ist. Die prozessalgebraische Darstellung der parallelen Komposition ist durch das Symbol $\mathcal{P}$ gegeben.

Die *Iteration (Loop)* ist im Gegensatz zu den vorangegangenen Operatoren ein einstelliger Operator, der die wiederholte Ausführung des enthaltenen Blocks im Sinne einer *while*-Schleife spezifiziert. Vergleichbar mit den Bedingungen der alternativen Komposition kann eine Abbruchbedingung für die Iteration angegeben werden, für die weder in der prozessalgebraischen noch in der graphischen Repräsentation linearer Prozessmodelle eine Darstellung erfolgt. Der Iterationsoperator wird in der Prozessalgebra durch den Präfixoperator $\mathcal{L}$ symbolisiert.

Jedes lineare Prozessmodell wird durch zwei spezielle Symbole (Prozessbegrenzer) eingeleitet bzw. abgeschlossen. Abbildung 2.10 zeigt die graphische Darstellung dieser Prozessbegrenzer sowie der Operanden und Operatoren linearer Prozessmodelle. Als Beispiel

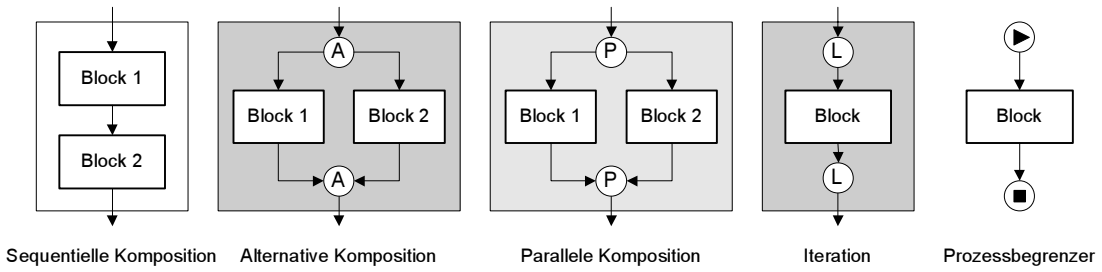


Abbildung 2.10: Beschreibungselemente eines linearen Prozessmodells

für ein lineares Prozessmodell zeigt Abbildung 2.11 den bereits bekannten Ausschnitt aus dem Geschäftsprozess „Güterbeschaffung“.

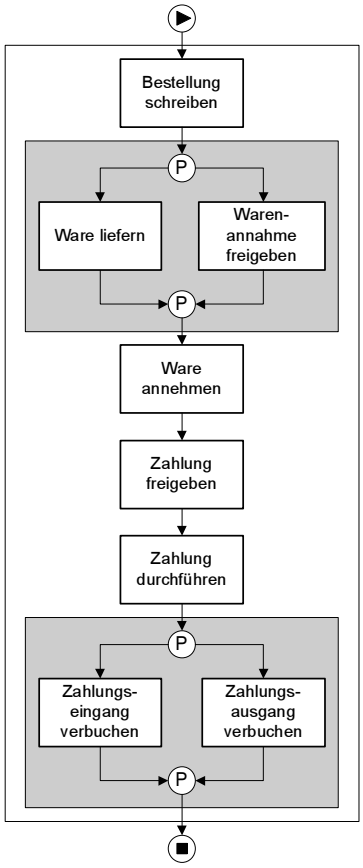


Abbildung 2.11: Beispiel eines linearen Prozessmodells (modelliert nach [FS96a])

Lineare Prozessmodelle weisen mit der ihnen zugrunde liegenden Prozessalgebra ein formales Fundament für ihre maschinelle Verarbeitung auf. Die Prozessalgebra definiert eine Menge von Axiomen, die in Tabelle 2.1 zusammengefasst sind und Eigenschaften wie Kommutativität, Assoziativität, Distributivität und Idempotenz ihrer Operatoren beschreiben [Sch03]. So sind die alternative und parallele Komposition kommutativ und somit unabhängig von der Reihenfolge ihrer Operanden (Axiome $A1$, $A6$). Für alle Operatoren der Prozessalgebra gilt Assoziativität, d. h. die Verschachtelung gleicher Operatoren kann aufgelöst werden (Axiome $A2$, $A5$ und $A8$). Im Hinblick auf den Einsatz eines sechsstelligen Sequenzoperators in Abbildung 2.11 soll ergänzend angemerkt werden, dass die Assoziativität die Verallgemeinerung der als zweistellige Operatoren eingeführten alternativen, sequentiellen und parallelen Komposition auf n-stellige Operatoren gestattet. Die Axiome zur Links-Distributivität der sequentiellen über die alternative Komposition (Axiom $A4$) sowie zur vollen Distributivität der parallelen über die alternative Komposition (Axiom $A7$ in Verbindung mit $A6$) beschreiben die Möglichkeit, einen Operator über seine Operanden zu verteilen. Die alternative Komposition ist als einziger Kompositions-

operator idempotent (Axiom $A3$), d. h. ihre Anwendung auf beliebige gleiche Operanden ergibt eben diesen Operanden.

$$
\begin{aligned}
A1: &\quad \mathcal{A}_1(x_1,\ldots,x_n) = \cdots = \mathcal{A}_{n!}(x_n,\ldots,x_1)\\
A2: &\quad \mathcal{A}(\ldots,x_1,\ldots,x_n,\ldots) = \mathcal{A}(\ldots,\mathcal{A}(x_1,\ldots,x_n),\ldots)\\
A3: &\quad \mathcal{A}(x,x,\ldots) = x\\
A4: &\quad \mathcal{S}(\mathcal{A}(x_1,\ldots,x_n),y_1,\ldots,y_m) = \mathcal{A}(\mathcal{S}_1(x_1,y_1,\ldots,y_m),\ldots,\mathcal{S}_n(x_n,y_1,\ldots,y_m))\\
A5: &\quad \mathcal{S}(\ldots,x_1,\ldots,x_n,\ldots) = \mathcal{S}(\ldots,\mathcal{S}(x_1,\ldots,x_n),\ldots)\\
A6: &\quad \mathcal{P}_1(x_1,\ldots,x_n) = \cdots = \mathcal{P}_{n!}(x_n,\ldots,x_1)\\
A7: &\quad \mathcal{P}(\mathcal{A}(x_1,\ldots,x_n),y_1,\ldots,y_m) = \mathcal{A}(\mathcal{P}_1(x_1,y_1,\ldots,y_m),\ldots,\mathcal{P}_n(x_n,y_1,\ldots,y_m))\\
A8: &\quad \mathcal{P}(\ldots,x_1,\ldots,x_n,\ldots) = \mathcal{P}(\ldots,\mathcal{P}(x_1,\ldots,x_n),\ldots)
\end{aligned}
$$

Tabelle 2.1: Axiome der Prozessalgebra linearer Prozessmodelle

Die Veröffentlichung einer formalen (operationellen) Semantik linearer Prozessmodelle steht zum Zeitpunkt der Anfertigung dieser Arbeit noch aus. Bisherige Publikationen zur linearen Prozessmodellierung haben sich auf die Angabe einer informellen Semantik beschränkt.

2.3 Zusammenfassung

Gegenstand dieses Kapitels war eine Einführung in die Grundlagen der Geschäftsprozessmodellierung. Dabei haben wir im Anschluss an die Definition der zentralen Begriffe dieses Gebiets die wichtigsten Anwendungsgebiete der Geschäftsprozessmodellierung vorgestellt, von denen für die geschäftsprozessorientierte Komponentensuche insbesondere die Einsatzmöglichkeiten im Rahmen der Anforderungsermittlung und -modellierung für die strategische Informationssystemplanung sowie die Auswahl und Entwicklung von (Standard-)Software relevant erscheinen. Um ein besseres Verständnis der Ausdrucksmächtigkeit aktueller Sprachen zur Geschäftsprozessmodellierung zu erhalten, haben wir mit EPKs, Aktivitätsdiagrammen und linearen Prozessmodellen drei Geschäftsprozessmodellierungssprachen eingehender vorgestellt. Auf eine Betrachtung der primär im Workflow-Umfeld eingesetzten Petri-Netze haben wir aufgrund ihres formalen Charakters verzichtet, obgleich die Anwendbarkeit höherer Petri-Netze zur Geschäftsprozessmodellierung z. B. mit den FUNSOFT-Netzen [DG98] gezeigt wurde.

Tabelle 2.2 stellt die von uns vorgestellten Sprachen vergleichend gegenüber und betrachtet dabei die Möglichkeiten zur Repräsentation betrieblicher Aktivitäten durch Vorgänge, deren organisatorische Zuordnung und den Zugriff auf Ressourcen einerseits sowie des Kontrollflusses durch syntaktische Konstrukte für Entscheidungen, parallele Abläufe und Schleifen andererseits. Darüber hinaus werden auch die Verständlichkeit der Sprachen für Fachexperten sowie deren formale Fundierung bewertet.

EPKs und Aktivitätsdiagramme bieten anders als lineare Prozessmodelle kein spezifisches Konstrukt für die Formulierung von Schleifen an. Da die Einhaltung von Modellierungsrichtlinien nicht bereits durch das Metamodell erzwungen wird, besteht die Gefahr der Erstellung nicht wohlgeformter Geschäftsprozessmodelle, die hinsichtlich der

Kriterium	EPK	Aktivitäts-diagramm	Lineares Prozessmodell
Aktivitäten			
Vorgang	+	+	+
Organisationseinheiten	+	+	∘
Ressourcen	+	+	∘
Kontrollfluss			
Entscheidungen	+	+	+
Parallelität	+	+	+
Schleifen	∘	∘	+
Sonstige Eigenschaften			
Wohlgeformtheit	∘	∘	+
Formalisierung	∘	∘	+
Verständlichkeit	+	+	∘
Verbreitung	+	+	−

Tabelle 2.2: Vergleich von EPK, Aktivitätsdiagramm und linearem Prozessmodell

Verwendung von Kontrollflussoperatoren keine korrekte Blockstruktur aufweisen und daher z. B. Deadlocks beinhalten können. Darüber hinaus verfügen sowohl EPKs als auch Aktivitätsdiagramme über keine solide formale Grundlage. Sie haben jedoch nicht zuletzt aufgrund ihrer intuitiven Verständlichkeit eine weite Verbreitung gefunden. Lineare Prozessmodelle basieren zwar mit einer Prozessalgebra auf einem soliden formalen Fundament und erzwingen die Erstellung wohlgeformter Modelle, sie sind jedoch aufgrund der nur implizit vorhandenen Repräsentation der organisatorischen Zuordnung von Vorgängen und des Zugriffs auf Ressourcen nur eingeschränkt verständlich. Da es sich bei linearen Prozessmodellen um einen eher forschungsorientierten Ansatz handelt, ist ihre Verbreitung vergleichsweise gering.

Trotz der Unterschiede zwischen den hier betrachteten Sprachen wird deutlich, dass Geschäftsprozessmodellierungssprachen einen gemeinsamen sprachlichen Kern[3] aufweisen, der Konstrukte für die Modellierung betrieblicher Vorgänge, deren organisatorische Zuordnung und Ressourcenzugriff sowie für die Spezifikation des Kontrollflusses durch Entscheidungen, Parallelität und Schleifen umfasst (vgl. hierzu auch [Gad02]).

[3] Wir verzichten hier bewusst auf den Begriff „Metamodell“, da z. B. in Petri-Netzen [Rei85] ein anderer Begriff des Prozessschritts als in EPKs, Aktivitätsdiagrammen oder linearen Prozessmodellen verwendet wird. In Petri-Netzen entspricht ein Prozessschritt einer globalen Transition, die durch die parallele Ausführung einer Menge gleichzeitig aktivierter (lokaler) Transitionen „schaltet“ [Old91].

Kapitel 3

Komponentensoftware

In diesem Kapitel legen wir die aus Sicht dieser Arbeit relevanten Aspekte der Komponentensoftware dar. Im Anschluss an die Einführung zentraler Grundbegriffe in Abschnitt 3.1 skizzieren wir in Abschnitt 3.2 den Makroprozess der komponentenbasierten Softwareentwicklung. Abschnitt 3.3 analysiert verschiedene Komponentenmodelle aus Forschung und Praxis hinsichtlich der Anforderungen an eine Komponentenbeschreibungssprache. Existierende Ansätze zur Beschreibung von Komponenten bewerten wir in Abschnitt 3.4, bevor wir das Kapitel mit einer Zusammenfassung in Abschnitt 3.5 beschließen.

3.1 Grundbegriffe

Der Begriff der Komponentensoftware bezeichnet den Gedanken der Konstruktion von Softwaresystemen durch die Kombination von (vorgefertigten) Komponenten, der seinen Ursprung bereits in der Softwarekrise der späten 60er Jahre hatte [McI69]. Seinerzeit war die Einführung eines Komponentenbegriffs in die Softwareentwicklung primär durch den Wunsch nach erhöhter *Wiederverwendung* [Kru92] zur Verbesserung von Software und deren Entwicklungsprozessen durch Qualitäts- und Produktivitätssteigerungen sowie Kostenreduktion getrieben. Biggerstaff und Perlis verstehen unter Wiederverwendung die »[...] wiederholte Anwendung verschiedenartigen Wissens über ein System auf ein anderes, ähnliches System mit dem Ziel, den Entwicklungs- und Wartungsaufwand des anderen Systems zu reduzieren.« (übersetzt nach [BP89]). Dabei beinhaltet das wiederverwendete Wissen u. a. Domänenwissen, Entwürfe, Architekturen, Anforderungen, Quellcode und Dokumentation.

Heute sind die Gründe für die Entwicklung von Komponentensoftware vielschichtiger. Neben dem vordergründigen Wunsch nach Wiederverwendung haben technische Fragestellungen (z. B. bzgl. Verteilung, Persistenz und Sicherheit), Flexibilitätsanforderungen, Interoperabilitätsaspekte, aber auch Fragen des Projekt- und Risikomanagements von Softwareprojekten höhere Priorität erlangt. Der *Komponentensoftware* in ihrer heutigen Form liegt daher die Idealvorstellung zugrunde, die Vorteile von Individual- und Standardsoftware miteinander zu kombinieren, indem Software durch die kundenindividuelle Komposition und Anpassung technisch und inhaltlich standardisierter Komponenten entwickelt wird [Szy02].

Obwohl die Idee der Komponentensoftware nunmehr über 30 Jahre alt ist und Komponententechnologien wie COM+ und Enterprise JavaBeans (EJB) bereits seit mehreren Jahren produktiv in der Softwareentwicklung eingesetzt werden, existiert bislang noch keine einheitliche Definition des Begriffs der Komponente in der Softwareentwicklung (vgl. z. B. [ND95, OHE96, Sam97, DW98, HS00, OMG03, ABC+02]). Die wohl größte Anerkennung hat die 1996 im Rahmen eines ECOOP-Workshops formulierte Definition erfahren, an deren überarbeitete Fassung [Szy02] wir uns mit unserem Komponentenbegriff im Rahmen dieser Arbeit anlehnen:

Definition 3.1 (Komponente) *Eine* Komponente *ist eine Abstraktions- und Kompositionseinheit, die eine in sich geschlossene, vermarktbare Lösung für einen abgegrenzten Problembereich anbietet. Komponenten werden als ausführbare Einheiten zur Verfügung gestellt, die aufgrund expliziter Spezifikationen von Schnittstellen und Kontextabhängigkeiten im Sinne einer Black-Box-Wiederverwendung durch Dritte genutzt werden können.*

Komponenten sind Elemente der Softwareentwicklung, die erst durch das Einbinden in eine Laufzeitumgebung (*Deployment*), wie sie z. B. ein Application Server darstellt, genutzt werden können. Sie weisen im Allgemeinen eine relativ grobe Granularität auf, d. h. sie umfassen – sofern eine objektorientierte Implementierung vorliegt – in der Regel mehr als eine Klasse. Die geforderten Spezifikationen von Schnittstellen und Kontextabhängigkeiten werden unter dem Begriff der *Komponentenbeschreibung* zusammengefasst:

Definition 3.2 (Komponentenbeschreibung, Komponentenbeschreibungssprache) *Eine* Komponentenbeschreibung *ist eine explizite Spezifikation der Schnittstellen und Kontextabhängigkeiten einer Komponente, die sowohl auf syntaktischer (technischer) als auch auf semantischer (fachlicher) Ebene formuliert ist und darüber hinaus auch die Komposition von Komponenten dokumentiert. Eine* Komponentenbeschreibungssprache *ist eine Sprache zur Erstellung von Komponentenbeschreibungen.*

Der Begriff der Komponente lässt sich weiter in GUI-, System- und Fachkomponenten differenzieren. *GUI-Komponenten* sind visuelle Komponenten, die der Konstruktion graphischer Benutzungsoberflächen (graphical user interface, GUI) dienen und folglich auf Clientseite eingesetzt werden. Im Gegensatz zu *Systemkomponenten*, die „horizontale“, also vom betrachteten Anwendungsbereich unabhängige Dienste bereitstellen, bieten *Fachkomponenten* „vertikale“ Dienste an, die eine fachliche, z. B. betriebswirtschaftliche Anwendungslogik implementieren. Abhängig vom primären Einsatzgebiet einer Fachkomponente lassen sich des Weiteren *Geschäftsobjektkomponenten* und *Prozesskomponenten* unterscheiden. Während Geschäftsobjektkomponenten den Zugriff auf persistent gehaltene Daten (*Geschäftsobjekte*) wie z. B. Aufträge oder Kunden ermöglichen, stellen Prozesskomponenten oftmals komplexere, fachlich relevante Dienste für die Abwicklung von Geschäftsprozessen wie z. B. die Auftragsbearbeitung bereit. Sowohl System- als auch Fachkomponenten können client- und serverseitig eingesetzt werden. Im Rahmen dieser Arbeit konzentrieren wir uns jedoch auf serverseitige Fachkomponenten, wie sie in modernen betrieblichen Informationssystemen verbreitet Anwendung finden, und bezeichnen diese verkürzend als Komponenten.

Um Ziele der Komponentensoftware wie Austauschbarkeit und Interoperabilität von Komponenten zu erreichen, ist es erforderlich, dass sich Komponenten an gewisse strukturelle und verhaltensorientierte Rahmenbedingungen halten. Solche Rahmenbedingungen werden in einem *Komponentenmodell* zusammengefasst (vgl. [GT00]):

Definition 3.3 (Komponentenmodell) *Ein* Komponentenmodell *spezifiziert einen Rahmen für die Entwicklung und den Einsatz von Komponenten, der strukturelle Anforderungen hinsichtlich der Komposition sowie verhaltensorientierte Anforderungen hinsichtlich der Kollaboration von Komponenten definiert. Ein Komponentenmodell stellt darüber hinaus Anforderungen an eine Infrastruktur für den Einsatz von Komponenten, zu denen verbreitet die Bereitstellung von Mechanismen für Verteilung, Persistenz, Nachrichtenaustausch, Sicherheit und Versionierung gehören.*

Zu den strukturellen Anforderungen eines Komponentenmodells gehört z. B. die Einhaltung vorgegebener Schnittstellenstandards. Verhaltensorientierte Anforderungen bestehen z. B. darin, dass vorgegebene Interaktionsprotokolle für die Kommunikation zwischen Infrastruktur und Komponente eingehalten werden. Eine konkrete Spezifikation einer Infrastruktur für den Einsatz von Komponenten wie z. B. COM+ und EJB bezeichnen wir als *Komponententechnologie*. Der Begriff des *Application Servers* wird vielfach synonym zur Bezeichnung eines Softwareprodukts, das eine Komponententechnologie implementiert, und gleichzeitig zur Bezeichnung einer konkreten Installation eines solchen Softwareprodukts auf einem zentral verfügbaren Rechner verwendet.

Komponenten können nach Szyperski [Szy02] grundsätzlich zwei Arten von Schnittstellen anbieten. *Direkte Schnittstellen* entsprechen prozeduralen Schnittstellen, wie sie in klassischen Softwarebibliotheken zu finden sind. *Indirekte Schnittstellen* dagegen sind Schnittstellen im objektorientierten Sinne, die von den Klassen einer Komponente implementiert und durch deren Instanzen (die Geschäftsobjekte) bereitgestellt werden. Obgleich aktuelle Komponententechnologien eine objektorientierte Implementierung von Komponenten fördern oder gar verlangen, gestattet der abstrakte Begriff der Komponente grundsätzlich die freie Wahl der Implementierungstechnologie. Herzum und Sims [HS00] unterscheiden hinsichtlich der Architektur von Komponenten zwei Stile. Komponenten des *typbasierten* (oder *servicebasierten*) Stils verbergen die Existenz von Geschäftsobjekten hinter einer direkten Schnittstelle, über die Geschäftsobjekte verwaltet, d. h. erzeugt, gesucht oder gelöscht werden können, und über die ebenfalls auf deren Eigenschaften zugegriffen werden kann. Beim *instanzbasierten* Architekturstil tritt eine Komponente nur in Gestalt ihrer Geschäftsobjekte nach außen, die sich über eigene Schnittstellen (die indirekten Schnittstellen der Komponente) ansprechen lassen. In vielen Komponentenmodellen ist eine Kombination des typbasierten Stils mit dem instanzbasierten Stil vorzufinden, die die explizite Unterscheidung der Verwaltung von Geschäftsobjekten vom Zugriff auf deren Eigenschaften gestattet.

Die Entwicklung von Anwendungssoftware auf der Grundlage von Komponenten folgt vielfach dem Mitte der 90er Jahre in der objektorientierten Programmierung populär gewordenen Konzept des *Frameworks* [Mat96]:

Definition 3.4 (Framework) *Ein* Framework *ist eine wiederverwendbare, „halbfertige“ Anwendung, die Struktur und Verhalten eines Entwurfs vorgibt und an so genannten* Hot

Spots *erweitert werden muss, um eine fertige, kundenindividuell angepasste Anwendung zu entwickeln (vgl. auch die Definitionen in [FS97, Joh97]).*

Im Gegensatz zu Klassenbibliotheken, die zur Laufzeit in den Kontrollfluss des kundenspezifischen Anwendungscodes eingebunden werden, definiert ein Framework den Kontrollfluss einer Anwendung selbst. Das Framework übernimmt als zentrale Instanz die Kontrolle über den Ablauf der Anwendung, indem es die kundenspezifischen Erweiterungen durch den Aufruf von Methoden ansteuert (*inversion of control*). Hinsichtlich der für die Erweiterung eines Frameworks eingesetzten Technik lassen sich *objektorientierte* oder *Klassenframeworks* (*White-Box-Frameworks*) und *Komponentenframeworks* (*Black-Box-Frameworks*) unterscheiden [FS97, Szy02]. Während Klassenframeworks mittels Vererbung angepasst werden, indem kundenspezifische Klassen von (abstrakten) Klassen des Frameworks abgeleitet werden, definieren Komponentenframeworks so genannte *Erweiterungspunkte*, an denen sie durch die Komposition mit kundenspezifischen Komponenten erweitert (parametrisiert) werden können. Dabei kann ein derart parametrisiertes Komponentenframework selbst eine Komponente darstellen, die in ein anderes Komponentenframework eingesetzt werden kann. Herzum und Sims unterscheiden diesbezüglich zwischen *kontinuierlich rekursiven* Komponentenmodellen, bei denen auf allen Stufen einer Kompositionshierarchie derselbe Komponentenbegriff verwendet wird, und *diskret rekursiven* Komponentenmodellen, bei denen jeder Hierarchiestufe ein spezifischer Komponentenbegriff zugeordnet ist [HS00]. Die Erweiterung eines Komponentenframeworks mit Komponenten wird kontrolliert, indem die Erweiterungspunkte mit *Kontrakten* typisiert werden.

Definition 3.5 (Kontrakt) *Ein* Kontrakt *stellt eine vertragsähnliche Vereinbarung zwischen einem Anbieter eines Dienstes (einer Komponente) und dessen Nutzer (einem Komponentenframework) dar, die neben funktionalen Aspekten auch nicht-funktionale Aspekte des Dienstes beinhalten kann (vgl. auch [Szy02]).*

Ein Komponentenframework darf an seinen Erweiterungspunkten nur mit Komponenten parametrisiert werden, die den jeweiligen Kontrakt erfüllen. Zu den funktionalen Aspekten eines Kontrakts gehören z. B. Syntax und Semantik der erwarteten Schnittstellen. Nicht-funktionale Aspekte beziehen sich z. B. auf die Zusicherung bestimmter Qualitätseigenschaften.

Wir sind nun in der Lage, die Entwicklung und Anpassung von Komponentensoftware begrifflich zu erfassen:

Definition 3.6 (Komponentenbasierte Softwareentwicklung) *Der Begriff der* komponentenbasierten Softwareentwicklung *bezeichnet die Entwicklung bzw. Anpassung von Komponentensoftware durch die Komposition von Komponenten auf der Grundlage von Komponentenframeworks. Dabei können neben neu entwickelten Komponenten auch vorgefertigte Komponenten zum Einsatz kommen, die möglicherweise von unterschiedlichen Herstellern stammen.*

Bei der komponentenbasierten Softwareentwicklung wird im Allgemeinen eine mehrschichtige Anwendungsarchitektur angestrebt, die auf die aufbau- und ablauforganisatorischen Anforderungen der zu unterstützenden Geschäftsprozesse ausgerichtet ist. Dabei

kommen serverseitig auf der obersten Schicht dieser Anwendungsarchitektur in der Regel Prozesskomponenten zur Unterstützung der Ausführung von Geschäftsprozessen zum Einsatz. Diese Prozesskomponenten greifen oftmals auf andere Prozesskomponenten oder aber Geschäftsobjektkomponenten zu, die auf darunter liegenden Schichten angesiedelt sind. Der direkte Einsatz von Geschäftsobjektkomponenten in der obersten Schicht ist dagegen eher selten zu beobachten.

3.2 Makroprozess komponentenbasierter Softwareentwicklung

Der Begriff der komponentenbasierten Softwareentwicklung umfasst neben technischen und fachlichen Aspekten wie z. B. der eingesetzten Komponententechnologie oder der Interoperabilität von Komponenten auch organisatorische Gesichtspunkte wie z. B. den Erwerb vorgefertigter Komponenten. Der Begriff des *Makroprozesses* konzentriert sich auf eben diese organisatorische Facette und beschreibt die an komponentenbasierten Softwareentwicklungsprozessen beteiligten Akteure sowie deren Aufgaben, Koordination und Kollaboration.

Charakteristisch für die komponentenbasierte Softwareentwicklung ist die Veränderung der Entwicklungsprozesse gegenüber traditionellen Softwareentwicklungsansätzen: Statt ein vorliegendes Problem lediglich in einem Top-Down-Ansatz zu partitionieren und eine Lösung z. B. in Form von Komponenten zu spezifizieren und implementieren, werden in den frühen Phasen komponentenbasierter Softwareentwicklungsprozesse neben verfügbaren Alt-Systemen (*Legacy-Systemen*) insbesondere auch auf einem Softwaremarkt angebotene Komponenten im Sinne einer ergänzenden Bottom-Up-Vorgehensweise berücksichtigt [Has02]. Dabei ist die Verwendung der Marktmetapher grundsätzlich unabhängig davon, ob Komponenten innerhalb einer Organisation oder zwischen verschiedenen Organisationen wiederverwendet werden sollen. In beiden Konstellationen treten Akteure in vergleichbaren Rollen auf, Unterschiede lassen sich jedoch hinsichtlich der Motivation dieser Akteure ausmachen. Während auf organisationsübergreifenden Softwaremärkten im Allgemeinen monetäre Anreize die Handelstätigkeit bestimmen, müssen auf organisationsinternen Marktplätzen künstliche Anreize wie z. B. Punktesysteme geschaffen werden, mit denen sich die Entwicklung wiederverwendbarer Komponenten einerseits sowie die Wiederverwendung angebotener Komponenten andererseits durch einen Entwickler positiv bewerten lässt. So berichtet Tracz in [Tra88], dass bei den Bell Laboratories der Titel „thief of the week“ an den Mitarbeiter verliehen wurde, der das höchste Maß an Wiederverwendung gezeigt hat. Biggerstaff und Richter [BR89] schlagen allgemein vor, Wiederverwendung zu belohnen, um das „not invented here“-Problem zu lösen, das der Wiederverwendung fremdentwickelter Komponenten entgegensteht. Einen Sonderfall stellen Marktplätze für Open-Source-Software wie z. B. *SourceForge.net* dar, bei denen Anreize wie Verbreitung von Software, Stabilisierung und Erweiterung der Software durch andere Entwickler, Prestigedenken einzelner Entwickler sowie die Möglichkeit zum Aufbau kommerzieller Softwaresysteme auf Basis von Open-Source-Software bestehen.

Ein Softwaremarkt für Komponenten, den wir im Folgenden als *Komponentenmarkt* [Kau00a, Szy02] bezeichnen, stellt eine elektronische Integrationsplattform für die ange-

botenen „Güter“, die Marktteilnehmer sowie verfügbare Dienstleistungen und somit die Grundlage des Makroprozesses der komponentenbasierten Softwareentwicklung dar. Die Güter eines Komponentenmarktes sind – wie der Name nahe legt – Komponenten, die gemeinsam mit ihren Komponentenbeschreibungen in einem *Komponenten-Repository* verwaltet werden. Das Schema dieses Repositories und damit die Komponentenbeschreibungssprache wird durch den *Marktplatzbetreiber* vorgegeben. Der Marktplatzbetreiber kann darüber hinaus optional einfache Such- und Navigationsfunktionalität im Repository anbieten, indem er z. B. Thesauri oder Kategorien definiert.

(Komponenten-)Hersteller bieten die von ihnen entwickelten Komponenten auf einem Komponentenmarkt an, indem sie diese in einem solchen Repository veröffentlichen. *(Komponenten-)Verwender* müssen im Rahmen der komponentenbasierten Softwareentwicklung die so genannte *Make-or-Buy-Entscheidung* treffen und dabei entscheiden, ob eine für ihre Anforderungen geeignete Komponente auf einem Komponentenmarkt verfügbar ist und erworben werden soll, oder ob eine Komponente mit entsprechender Funktionalität selbst zu entwickeln ist. Ein *(Komponenten-)Broker* unterstützt Verwender bei dieser Aufgabe durch Suchdienste, die auf der Grundlage der Anforderungen eines Verwenders geeignete Komponenten ermitteln. Der Erwerb einer Komponente durch einen Verwender kann je nach Marktplatz wahlweise durch den Broker in Verbindung mit dem Marktplatzbetreiber oder aber direkt mit deren Hersteller abgewickelt werden.

Weitere Dienstleistungen, die neben der Komponentensuche auf einem Komponentenmarkt angeboten werden können, sind z. B. die Möglichkeit zur Versionierung eingestellter Komponenten, die Benachrichtigung von Verwendern einer Komponente bei Erscheinen einer neuen Version oder die Option, neben Ad-hoc-Anfragen auch Suchanfragen zu formulieren, die über einen längeren Zeitraum gültig bleiben. Abbildung 3.1 skizziert den beschriebenen Makroprozess der komponentenbasierten Softwareentwicklung.

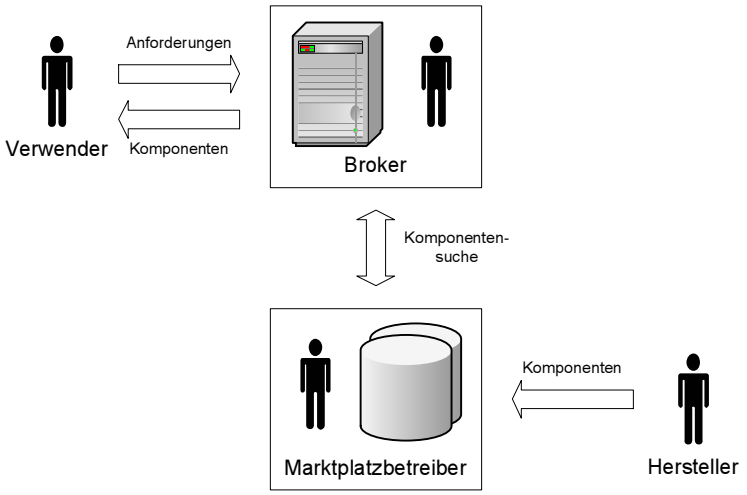


Abbildung 3.1: Makroprozess der komponentenbasierten Softwareentwicklung

Neben diesem generischen Makroprozess existieren noch verschiedene Vorschläge für differenziertere Modelle. So berücksichtigen Sandmann et al. in [STR00] u. a. die Rolle

des *Anwendungsarchitekten*, der auf der Grundlage von Komponentenframeworks aus einfachen, oftmals domänenunabhängigen Komponenten komplexere Komponenten mit anwendungsnahem Charakter entwickelt. Der *Konfigurierer* (auch als Berater bezeichnet) berät den Verwender und unterstützt ihn z. B. bei der Auswahl von Komponenten. Zudem zeigt er dem Verwender Konfigurationsalternativen ausgewählter, aber noch unkonfigurierter (komplexer) Komponenten auf, bewertet diese und konfiguriert die Komponenten entsprechend. Kaufmann fasst diese Rollen in seinem Markt für heterogene Komponenten betrieblicher Anwendungssysteme [Kau00b] in der Rolle des *(Komponenten-)Integrators* zusammen, der als Intermediär zwischen Verwender und Marktplatzbetreiber bzw. Komponenten-Repository auftritt. Die Aufgabe des Integrators ist es, anhand der ermittelten Anforderungen des Kunden geeignete Komponenten auf dem Marktplatz zu suchen und diese dann in das Gesamtsystem zu integrieren. Vergleichbare und weitere Vorschläge für Rollen von Teilnehmern auf einem Komponentenmarkt finden sich auch bei Karlsson [Kar95], Sawyer [Saw01] und Szyperski [Szy02].

Aktuelle internetbasierte Komponentenmärkte wie z. B. ComponentSource [Com03], Sun iForce Partner Products Catalog [Sun03], SoftSelect [Sof03], SAP Software Partner Directory [SAP03], Software-Marktplatz [Nom03] oder der Open-Source-Marktplatz SourceForge.net [Sou03] betrachten lediglich einen einfachen Makroprozess, der dem in Abbildung 3.1 dargestellten ähnelt, jedoch die Rolle des Brokers kaum berücksichtigt. Wir skizzieren die im Kontext der Komponentensuche relevanten Eigenschaften aktueller Komponentenmärkte in Abschnitt 4.2.

3.3 Komponentenmodelle

Gegenstand dieses Abschnitts ist die Vorstellung verschiedener Komponentenmodelle aus Forschung und Praxis. Dabei berücksichtigen wir lediglich solche Aspekte, die die Wahl bzw. den Entwurf einer von den Spezifika einzelner Komponentenmodelle unabhängigen Komponentenbeschreibungssprache maßgeblich beeinflussen können. Da wir im Rahmen dieser Arbeit nicht an den Eigenschaften der durch ein Komponentenmodell spezifizierten Infrastruktur für den Einsatz von Komponenten interessiert sind, verzichten wir auf eine Betrachtung der bereitgestellten Mechanismen für Verteilung, Persistenz, Nachrichtenaustausch, Sicherheit, Versionierung o. ä.

Im Folgenden stellen wir zunächst konzeptionelle Komponentenmodelle aus der Forschung vor, bevor wir auf die Komponentenmodelle aktueller Komponententechnologien eingehen.

3.3.1 Konzeptionelle Komponentenmodelle aus der Forschung

Wir betrachten mit dem REBOOT-, dem Component/Connector- und dem Business-Component-Komponentenmodell drei konzeptionelle Komponentenmodelle, denen eine unterschiedliche Zielsetzung zugrunde liegt. Während das REBOOT-Komponentenmodell auf den Entwurf von Komponenten-Repositories ausgerichtet ist, stellt das Component/Connector-Komponentenmodell eine ontologische Grundlage für die (formale) Beschreibung von komponentenbasierten Softwarearchitekturen dar. Das Business-Component-

Komponentenmodell schließlich definiert eine abstrakte Architektur für den Aufbau von Fachkomponenten.

REBOOT-Komponentenmodell Das Forschungsprojekt *REBOOT (REuse Based on Object-Oriented Techniques)* [Kar95] wurde zwischen 1993 und 1994 im Rahmen des ESPRIT-3-Programms mit dem Ziel durchgeführt, die Produktivität und Qualität in der Softwareentwicklung durch die Förderung und Unterstützung der organisierten Wiederverwendung zu erhöhen. Der Begriff der organisierten Wiederverwendung bezeichnet dabei die Verwendung eines expliziten Wiederverwendungsprozesses, der die zwei übergeordneten Aktivitäten „develop *for* reuse“ und „develop *with* reuse“ umfasst.

Die Schnittstelle zwischen diesen beiden Entwicklungsaktivitäten bildet ein Komponenten-Repository, dessen Schema durch das REBOOT-Komponentenmodell vorgegeben ist. Das REBOOT-Komponentenmodell stellt also weniger einen Rahmen für die Entwicklung von Komponenten dar, sondern vielmehr eine Struktur für deren umfassende Beschreibung. Abbildung 3.2 zeigt das REBOOT-Komponentenmodell in Form eines Klassendiagramms.

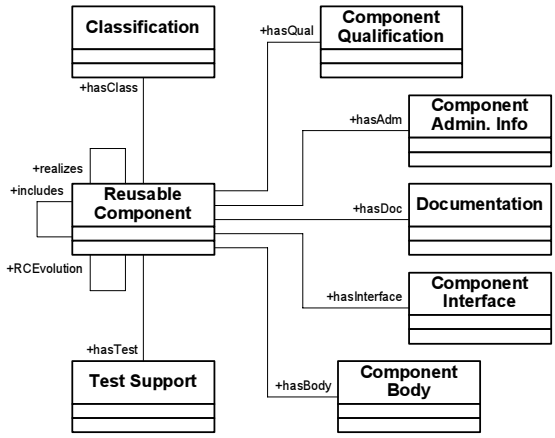


Abbildung 3.2: REBOOT-Komponentenmodell

In der REBOOT-Terminologie ist eine Komponente (*Reusable Component* im Klassendiagramm) ein abgeschlossenes Produkt auf einer bestimmten Entwicklungsstufe (Analyse, Entwurf, Implementierung etc.) zusammen mit den für seine Wiederverwendung erforderlichen Informationen. Eine Komponente kann zwecks Identifikation und Suche klassifiziert werden (*Classification*). Ihr können Informationen, die ihre Qualität und Wiederverwendbarkeit mittels einer Metrik bewerten, zugeordnet werden (*Component Qualification*). Für die Verwaltung von Komponenten können darüber hinaus noch allgemeine Informationen über die Komponente (z. B. Angaben zu ihrem Entwickler oder das Datum ihrer Veröffentlichung im Repository), Informationen über Berechtigungen zur Wiederverwendung der Komponente durch spezifische Entwicklergruppen sowie Preisangaben abgelegt werden (*Component Administrative Information*). Die Dokumentation einer Komponente soll dazu beitragen, ihre Eignung hinsichtlich gegebener Anforderungen bewerten, ihre angebotene Funktionalität verstehen und sie ggf. spezifisch anpassen zu

können (*Documentation*). Von besonderer Relevanz für den Einsatz einer Komponente sind deren Schnittstelle (*Component Interface*) und ihre Implementierung (*Component Body*). In diesem Zusammenhang ist anzumerken, dass eine Komponente im REBOOT-Komponentenmodell genau eine Schnittstelle besitzt. Der Test einer wiederverwendeten Komponente in einem spezifischen Kontext wird durch die Spezifikation von Testprozeduren, Testdaten sowie erwarteten Ergebnissen erleichtert (*Test Support*).

Die *realizes*-Beziehung zwischen Komponenten reflektiert die Annahme, dass eine Komponente auf verschiedenen Abstraktions- oder Entwicklungsstufen (z. B. Analyse, Design und Implementierung) im Repository existieren kann, die im Sinne einer Verfeinerungsbeziehung zusammenhängen (siehe auch Abschnitt 7.1). Mittels der *includes*-Beziehung wird die Komposition komplexer Komponenten aus einer Menge einfacherer Komponenten in Form von Verweisen auf diese Komponenten abgebildet. Das REBOOT-Komponentenmodell sieht dabei keine Kontrakte für die Beschreibung der Kompositionsmöglichkeiten von Komponenten vor. Die *RCEvolution*-Beziehung dient der Verfolgung der Versionshistorie von Komponenten.

Component/Connector-Komponentenmodell Shaw und Garlan identifizieren in ihrer Arbeit zum Entwurf von *Architekturbeschreibungssprachen (Architecture Description Languages, ADLs)* [SG96] mit *Komponenten (Components)* und *Konnektoren (Connectors)* zwei grundlegende Typen von Elementen, aus denen (Software-)Systeme aufgebaut sind. Die Unterscheidung dieser beiden Elementtypen gestattet die Trennung der Spezifikation einer Komponente von der Beschreibung ihrer Interaktionen im Rahmen ihrer tatsächlichen Nutzung.

Komponenten entsprechen in dieser Terminologie weitgehend den Übersetzungseinheiten einer konventionellen Programmiersprache. Sie sind Träger von Zustand und Verhalten, die Eigenschaften wie z. B. Signaturen, Funktionalität und Performance in ihrer Schnittstelle definieren. Dabei ist die Schnittstelle einer Komponente durch eine Menge so genannter *Anschlüsse (Ports)* definiert, die Interaktionspunkte mit der Komponente festlegen. Komponenten können entweder einfach oder zusammengesetzt sein.

Konnektoren dienen als Vermittler zwischen interagierenden Komponenten, d. h. sie sind die Träger der strukturellen und verhaltensorientierten Beziehungen zwischen Komponenten. Sie definieren ihre Schnittstelle in Form einer Menge von *Rollen (Roles)*, die spezifische, über den Konnektor verbundene Komponenten einnehmen müssen. Jedem Konnektor ist eine Protokollspezifikation zugeordnet, die seine Eigenschaften definiert. Dazu gehören – vergleichbar mit einem Kontrakt – Regeln bzgl. der Typen von Anschlüssen, zwischen denen der Konnektor vermitteln kann, Zusicherungen über die Eigenschaften der unterstützten Interaktionen bzw. Interaktionssequenzen sowie Regeln über die Reihenfolge einzelner Interaktionen. Ähnlich wie Komponenten können auch Konnektoren einfach oder zusammengesetzt sein.

Konnektoren stellen die Grundlage für die Komposition von Komponenten zu *zusammengesetzten Komponenten (Composite Components)* dar. Eine zusammengesetzte Komponente beschreibt eine gültige Konfiguration von (einfachen oder zusammengesetzten) Komponenten und Konnektoren, bei der die Anschlüsse der Komponenten auf die Rollen der Konnektoren abgebildet sind und zudem sichergestellt ist, dass die resultierende Komposition die Spezifikationen der Schnittstellen der beteiligten Komponenten und der

Protokolle der Konnektoren erfüllt. Abbildung 3.3 fasst das Component/Connector-Komponentenmodell graphisch zusammen.

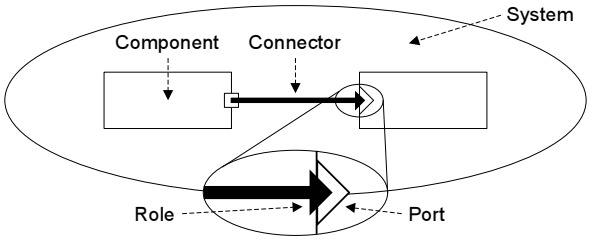


Abbildung 3.3: Component/Connector-Komponentenmodell (nach [GMW00])

Business-Component-Komponentenmodell Herzum und Sims schlagen mit dem *Business Component Approach* [HS00] ein umfassendes konzeptionelles Rahmenwerk für die Entwicklung verteilter betrieblicher Informationssysteme auf der Grundlage von Komponenten vor. Abbildung 3.4 stellt das diskret rekursive Komponentenmodell ihres Ansatzes in Form eines Klassendiagramms graphisch dar.

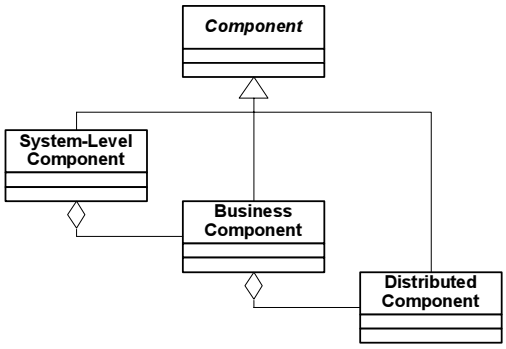


Abbildung 3.4: Business-Component-Komponentenmodell (nach [HS00])

Auf der untersten Granularitätsstufe finden sich *verteilte Komponenten (Distributed Components)*, die mit aktuellen Komponententechnologien wie z. B. COM und EJB implementiert werden. Eine verteilte Komponente hat eine präzise definierte Schnittstelle, ist unabhängig einsetzbar und über ein Netzwerk adressierbar.

Die *Fachkomponente (Business Component)* stellt als Implementierung eines autonomen betrieblichen Konzepts oder Prozesses die zentrale Abstraktion des Rahmenwerks dar. Da sie im Allgemeinen aus einer oder mehreren verteilten Komponenten besteht, kann eine Fachkomponente in einem verteilten Szenario eingesetzt werden. Diese Verteilung wird insbesondere dadurch unterstützt, dass eine Fachkomponente grundsätzlich aus vier logischen Schichten aufgebaut ist, die nicht notwendigerweise von jeder Fachkomponente vollständig implementiert werden müssen (siehe Abbildung 3.5):

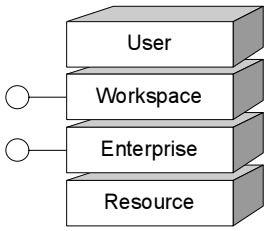


Abbildung 3.5: Vier logische Schichten einer Fachkomponente (nach [HS00])

- *Präsentationsschicht (User Tier)*: Die Aufgabe der Präsentationsschicht ist die graphische Repräsentation einer Fachkomponente zwecks Interaktion mit einem Benutzer.
- *Arbeitsplatzschicht (Workspace Tier)*: Die Arbeitsplatzschicht unterstützt die Präsentationsschicht durch die Implementierung benutzerspezifischer Geschäftslogik. Sie ist zudem verantwortlich für die Interaktion mit der Unternehmensschicht.
- *Unternehmensschicht (Enterprise Tier)*: Die Unternehmensschicht realisiert die zentralen fachlichen Aspekte einer Fachkomponente. Sie implementiert unternehmensweite Geschäftslogik und überwacht die Integrität der Geschäftsdaten.
- *Ressourcenschicht (Resource Tier)*: Aufgabe der Ressourcenschicht ist die Abbildung zwischen dem auf der Unternehmensschicht implementierten Modell und tatsächlich verfügbaren Ressourcen wie z. B. Datenbanken, die über ein Netzwerk verteilt vorliegen können.

Herzum und Sims unterscheiden neben den von uns betrachteten *Geschäftsobjektkomponenten (Entity Components)* und *Prozesskomponenten (Process Components)* noch zwei weitere Kategorien von Fachkomponenten. *Hilfskomponenten (Utility Components)* realisieren betriebliche Konzepte wie Kalender, Adressverwaltung oder Maßeinheiten, die in verschiedenen Kontexten Anwendung finden können. *Zusatzkomponenten (Auxiliary Components)* repräsentieren in der Regel kein betriebliches Konzept und sind z. B. für die Wahrung der Datenbankintegrität oder die Überwachung der Performance verantwortlich.

Eine Menge kooperierender Fachkomponenten, die gemeinsam einen oder mehrere eigenständige Geschäftsprozesse implementieren, wird *Fachkomponentensystem (Business Component System)* genannt. Bietet ein Fachkomponentensystem eine definierte Schnittstelle an, die ihre Wiederverwendung im Sinne einer *Black Box* gestattet, so wird sie als *System-Level Component* bezeichnet.

3.3.2 Komponentenmodelle aktueller Komponententechnologien

Wir stellen nun die Komponentenmodelle der Komponententechnologien Component Object Model (COM) und .NET, Enterprise Java Beans (EJB) sowie CORBA Component Model (CCM) übersichtsartig vor und diskutieren schließlich die verbreitete Einordnung von Web Services als neuer Komponententyp.

Component Object Model und .NET Microsoft hat in den 90er Jahren mit dem *Component Object Model (COM)* eine sehr erfolgreiche Komponententechnologie entwickelt, die – nach einer Reihe verschiedener Ergänzungen – erst in jüngster Zeit durch *.NET* ergänzt bzw. abgelöst wird.

COM [Bro95] stellt einen Standard für die Entwicklung von Komponenten dar, der Anforderungen an das binäre Format einer Komponente spezifiziert, die Wahl der Implementierungstechnologie (Paradigma, Programmiersprache) jedoch freistellt [Szy02]. Der grundlegende Elementtyp des Komponentenmodells von COM ist die (eingehende) *Schnittstelle (Interface)*, die eine Menge (indirekter) Verweise auf Operationen zusammenfasst. Eine *COM-Klasse (COM Class)* ist eine benannte Implementierung mindestens einer solchen Schnittstelle. Da COM als binärer Standard die Zuordnung implementierter Methoden zu angebotenen Schnittstellen auf technischer Ebene über Verweisstrukturen vornimmt, kann eine COM-Klasse verschiedene Implementierungen derselben Operation kapseln und unterschiedlichen Schnittstellen (z. B. einer Schnittstelle in verschiedenen Versionen) zuordnen.

Die Instanzen einer COM-Klasse werden als *COM-Objekte (COM Objects)* bezeichnet und mittels so genannter *Class Factories* erzeugt. Eine Class Factory (oder präziser ein Class Factory Object) ist ein spezielles COM-Objekt, das die Standard-Schnittstelle `IClassFactory` unterstützt und zwecks „bootstrapping“ von der COM-Bibliothek erzeugt wird. Ein *COM-Server* repräsentiert im COM-Komponentenmodell eine unabhängig einsetzbare Komponente. Er „verpackt“ mindestens eine COM-Klasse und implementiert für jede dieser Klassen eine entsprechende Class Factory. Obgleich die Class Factories eines COM-Servers als dessen direkte Schnittstelle verstanden werden können, ist der durch COM realisierte Architekturstil als instanzbasiert zu bezeichnen, d. h. Komponenten treten nur in Gestalt ihrer Geschäftsobjekte nach außen. Das COM-Komponentenmodell unterstützt keine Vererbung zwischen seinen Komponenten.

Da COM-Klassen die von ihnen unterstützten Schnittstellen nicht explizit bekannt geben, ist es im Allgemeinen notwendig, die Eignung einer COM-Klasse für einen bestimmten Anwendungskontext durch die individuelle Abfrage unterstützter Schnittstellen zu prüfen (*interface negotiation*). Um den bei einer großen Anzahl erwarteter Schnittstellen anfallenden Aufwand zu vermeiden, stellt COM neben den bereits erwähnten Schnittstellen das Konzept der *Kategorien (Categories)* zur Beschreibung des Typs einer COM-Klasse bereit. Eine Kategorie spezifiziert eine Menge von Schnittstellen und macht somit den Gedanken des Kontrakts explizit.

Abbildung 3.6 stellt die grundlegenden Konzepte von COM sowie deren Beziehungen untereinander differenziert nach Typ-, Implementierungs- und Laufzeitsicht auf eine Komponente dar. In der Abbildung wird o. B. d. A eine objektorientierte Implementierung der COM-Klassen angedeutet. Da COM lediglich einen binären Standard definiert und keinerlei Aussagen über die primär mit seinen Komponenten verfolgten Aufgaben macht, kann eine Klassifizierung gemäß der von uns betrachteten Einteilung in Prozess- und Geschäftsobjektkomponenten nicht vorgenommen werden.

Neben den bereits erwähnten *eingehenden* Schnittstellen bietet COM als Teil der *COM Services* auch so genannte *ausgehende* Schnittstellen an. Ausgehende Schnittstellen eines COM-Objekts können zum einen von anderen COM-Objekten genutzt werden, die in der Rolle eines *Beobachters* [GHJV01] über Ereignisse o. ä. benachrichtigt werden

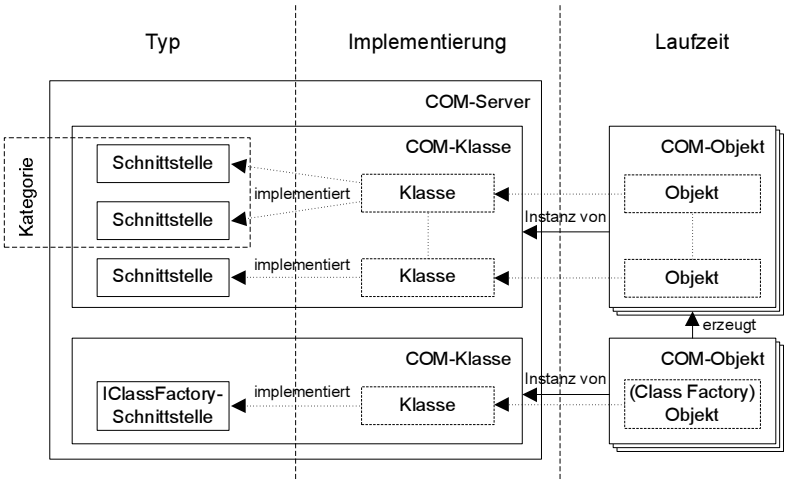


Abbildung 3.6: Komponentenmodell von Component Object Model (COM)

wollen, zum anderen können sie als Deklaration geforderter Schnittstellen bei der Komposition von COM-Objekten verstanden werden [Szy02]. Ausgehende Schnittstellen werden insbesondere von *ActiveX-Controls* [Cha96] genutzt. *ActiveX* ist eine Client-Technologie für die Entwicklung internetbasierter „Mini-Applikationen“, die aus den früheren *Visual Basic Controls* hervorgegangen ist und nunmehr auf COM basiert. COM hat darüber hinaus im Laufe der Jahre verschiedene Ergänzungen erfahren. So erweitert *Distributed COM (DCOM)* [OHE99] die Konzepte und Dienste von COM von der bereits in COM unterstützten Interprozesskommunikation auf den Einsatz in verteilten Rechnernetzen. *COM+* [OHE99] vereint als Nachfolger von DCOM bis dahin getrennte Technologien zur Unterstützung von Transaktionsverarbeitung, asynchronem Nachrichtenaustausch, Lastverteilung und Clustering. Zu den bekanntesten Vorläuferprodukten von COM+ gehören der *Microsoft Transaction Server* und der *Microsoft Message Queue Server* [Szy02].

Mit der *.NET-Initiative* [Szy02] als jüngste Entwicklung auf dem Gebiet der Komponententechnologien verfolgt Microsoft die Vision der Entwicklung heterogener Softwaresysteme in Form vernetzter Produkte und Dienste. .NET spezifiziert mit der *Common Language Infrastructure (CLI)* eine sprachneutrale Plattform, die – anders als CORBA (Common Object Request Broker Architecture) – eine Zwischensprache namens *Common Intermediate Language (CIL)* und ein Dateiformat für einsetzbare Softwaremodule definiert. Das *.NET-Framework* [Pro02] beinhaltet als Teil der .NET-Initiative

- die *Common Language Runtime (CLR)*, eine Implementierung der CIL,
- eine große Anzahl von Frameworks (z. B. für die Entwicklung von Web Services und die Verarbeitung von XML) sowie
- verschiedene Werkzeuge.

Die CLR unterstützt über die durch die CIL definierten Anforderungen hinaus u. a. die Interoperabilität gemäß COM+.

CIL-Programme, die sich durch die CLR ausführen lassen, werden in .NET als *verwaltete Module* bezeichnet. *Assemblies* sind unabhängig einsetzbare Einheiten und repräsentieren somit den Begriff der Komponente in .NET. Eine Assembly ist technisch betrachtet eine Menge von Dateien, die in einer Verzeichnishierarchie angeordnet sind, und enthält neben verwalteten Modulen auch Metadaten zur Beschreibung ihrer Schnittstellen sowie evtl. benötigte Ressourcen, die jeweils plattformunabhängig repräsentiert sind [Szy02]. Assemblies stellen (neben dem durch die CIL gesteckten Rahmen) keine spezifischen Anforderungen an die enthaltenen Module und definieren folglich kein strukturiertes Komponentenmodell wie dies z. B. bei COM der Fall ist.

Enterprise JavaBeans *Enterprise JavaBeans (EJB)* [DYK01] ist ein zentraler Baustein von Suns *Java 2 Enterprise Edition (J2EE)*, einer Sammlung von Spezifikationen für die plattformunabhängige Entwicklung betrieblicher Anwendungssoftware auf der Grundlage von Java. Die Komponenten des EJB-Komponentenmodells (die *Enterprise Beans* oder kurz *Beans*) werden zur Laufzeit in so genannten *Containern* verwaltet, die durch einen Application Server realisiert werden und ihnen als Teil ihrer operativen Einsatzumgebung Dienste wie Transaktions- und Sicherheitsmanagement, Verteilung in Netzwerken, Ressourcenverwaltung etc. transparent zur Verfügung stellen.

Neben diesen Diensten stellen Container Schnittstellen für die Entwicklung von Anwendungsclients und Enterprise Beans zur Verfügung, die in Form von Verträgen festgelegt werden. Der *Client Contract* regelt als Vertrag zwischen Client und Container den Zugriff eines Clients auf die in einem Container eingesetzten Beans. Dabei sind aus der Perspektive eines Clients zwei Arten von Enterprise Beans relevant. Während *Entity Beans* im Sinne von Geschäftsobjektkomponenten den Zugriff auf persistent gehaltene Daten kapseln, bieten *Session Beans* als Form von Prozesskomponenten (fachlich relevante) Dienste an, die Teil einer Folge von Client-Server-Interaktionen (einer „Session“) sein können. Session Beans können weiter nach ihrer Unterstützung zustandsbehafteter und zustandsloser Sessions in *Stateful* und *Stateless* Session Beans differenziert werden. Der Client Contract verlangt von Entity und Session Beans die Definition eines *Home Interfaces*, das als direkte Schnittstelle der Bean Methoden zur Erzeugung und (bei Entity Beans) zum Auffinden und Löschen von *Bean-Instanzen* anbietet. Seit Version 2.0 der EJB-Spezifikation kann das Home Interface einer Entity Bean zudem so genannte *Home (Business) Methods* umfassen, die den Zugriff auf Geschäftslogik ermöglichen, die nicht spezifisch für eine einzelne Bean-Instanz ist. Daneben müssen Entity und Session Beans als indirekte Schnittstelle ein *Component Interface* spezifizieren, über das Bean-Instanzen ihre Geschäftslogik bereitstellen. Da eine Enterprise Bean sowohl eine direkte als auch eine indirekte Schnittstelle definiert, realisiert EJB eine Kombination des instanzbasierten und des typbasierten Architekturstils.

Der *Component Contract* spezifiziert die Schnittstelle zwischen einem Container und einer darin enthaltenen Enterprise Bean. Er legt u. a. fest, dass der Container Implementierungen des Home und Component Interfaces bereitstellen (z. B. generieren) muss, die eine Enterprise Bean vom direkten Zugriff durch Clients abschirmen. Die Implementierung des Component Interfaces delegiert dabei Aufrufe eines Clients an die *Enterprise Bean Class*, die die Geschäftslogik einer Entity oder Session Bean definiert. Instanzen dieser Implementierungen werden als *EJBHome* respektive *EJBObject* bezeichnet und

fungieren als *Proxy*-Objekte [GHJV01] für die eigentliche Bean-Instanz (*interception*). Während genau ein EJBHome pro Entity oder Session Bean existiert, wird für jede Bean-Instanz ein EJBObject angelegt. Neben diesem Mechanismus zur Kapselung der Bean-Instanzen vom direkten Clientzugriff umfasst der Component Contract u. a. auch Vereinbarungen über den Lebenszyklus einer Enterprise Bean, die Verwaltung von Transaktionen und Sicherheit sowie die Bereitstellung von Persistenz- und Namensdiensten. Das EJB-Komponentenmodell unterstützt keine Vererbung zwischen Komponenten. Abbildung 3.7 fasst das EJB-Komponentenmodell graphisch zusammen.

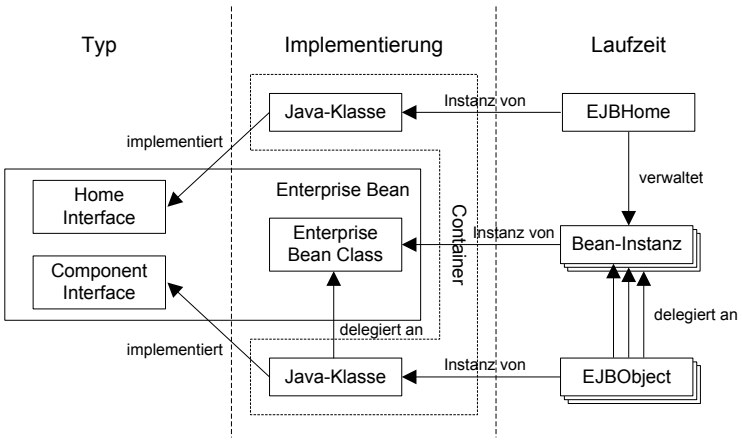


Abbildung 3.7: Komponentenmodell von Enterprise JavaBeans (EJB)

Entity Beans stellen objektorientierte Repräsentationen von Geschäftsobjekten wie z. B. Kunden und Aufträgen dar, zwischen denen auf Datenbankebene im Allgemeinen (Fremdschlüssel-)Beziehungen bestehen. Die EJB-Spezifikation berücksichtigt diesen Sachverhalt seit Version 2.0 durch die Ergänzung von *Relationships* zwischen Entity Beans, über die zwischen verbundenen Bean-Instanzen ähnlich wie über Fremdschlüsselbeziehungen auf Datenbankebene navigiert werden kann. Darüber hinaus werden *Message-Driven Beans* als weiterer Typ von Enterprise Beans eingeführt. Message-Driven Beans implementieren wie Entity und Session Beans eine serverseitige Geschäftslogik, verwenden aber ein abweichendes Kommunikationsmodell: An die Stelle direkter Aufrufe durch einen Client tritt der Eingang von Nachrichten, die über den *Java Message Service* an den Container übermittelt und der Message-Driven Bean aufgrund ihrer Registrierung für bestimmte Inhalte zugeordnet wurden. Im Gegensatz zu Entity und Session Beans definieren Message-Driven Beans daher weder ein Home Interface noch ein Component Interface. Message-Driven Beans verwalten wie zustandslose Session Beans keine Informationen über den Verlauf einer Sitzung (konversationaler Zustand) [Szy02].

CORBA Component Model Das *CORBA Component Model (CCM)* [OMG02a] ist ein plattform- und sprachunabhängiges Komponentenmodell der Object Management Group (OMG), das ausgewählte Konzepte von COM und EJB miteinander vereint. Es kann insbesondere als logische Erweiterung des EJB-Komponentenmodells betrachtet werden und verspricht die Kompatibilität mit Version 1.1 des EJB-Komponentenmodells.

Im CORBA Component Model beschreibt der Begriff des *Component Types* die Spezifikation einer Menge von Eigenschaften einer Komponente. Ein solcher Component Type kann verschiedene Implementierungen haben (z. B. für unterschiedliche Plattformen), die als *CCM Components* bezeichnet werden. Ähnlich wie bei EJB werden Instanzen von CCM Components durch *Homes* verwaltet, die im Unterschied zu EJB allerdings nicht Teil der Komponente sind. Die Schnittstelle eines Homes bietet Lebenszyklus-Methoden zum Erzeugen, Suchen und Löschen von Instanzen und kann darüber hinaus auch weitere Methoden im Sinne der durch EJB 2.0 unterstützten *Home (Business) Methods* enthalten. Instanzen einer CCM Component können durch mehrere unabhängige Homes verwaltet werden, wobei allerdings jede Instanz genau einem Home zugeordnet ist.

Eine CCM Component hat eine ausgezeichnete Schnittstelle, die als *Equivalent Interface* bezeichnet wird und alle Merkmale der Komponente exportiert. Zu diesen Merkmalen, die als *Ports* bezeichnet werden, gehören Facets, Receptacles, Event Sources und Sinks sowie Attributes. *Facets* stellen so genannte *provided interfaces* dar, die ähnlich wie Nicht-Standard-Schnittstellen in COM die Anwendungsschnittstellen einer CCM Component repräsentieren. Ein Client einer Instanz einer CCM Component kann – ähnlich wie bei COM – von jeder Facet zum Equivalent Interface navigieren, und umgekehrt kann ein Client vom Equivalent Interface jede Facet erhalten. Neben Facets kann eine CCM Component weitere, „einfache“ CORBA-Schnittstellen unterstützen. *Receptacles* stellen eine Abstraktion zur Erzeugung und Verwaltung getypter Verbindungen zwischen einer Instanz einer CCM Component und einem Objekt (z. B. einer Referenz auf eine Facet oder das Home einer anderen CCM Component) dar. Sie dienen damit der Repräsentation von Schnittstellen, die eine CCM Component zur Laufzeit von anderen Objekten erwartet (*required interfaces*). Das CORBA Component Model unterstützt die ereignisbasierte Kommunikation nach dem *Publish/Subscribe-Modell*. Dabei veröffentlichen *Event Sources* Ereignisse, die von registrierten „Verbrauchern“ über *Event Sinks* konsumiert werden. *Attributes* dienen vornehmlich der Konfiguration von Komponenten über Accessor-Methoden. Die Konfiguration von Komponenten wird im CORBA Component Model explizit durch die Einteilung des Lebenszyklus einer Komponente in eine Konfigurationsphase und eine operationelle Phase unterstützt.

CCM realisiert ähnlich wie EJB eine Kombination des instanzbasierten und des typbasierten Architekturstils, bei dem Komponenten sowohl über eine direkte Schnittstelle (Home) als auch über weitere indirekte Schnittstellen (Facets und konventionelle CORBA-Schnittstellen) angesprochen werden können. Abbildung 3.8 illustriert die wichtigsten Zusammenhänge des CORBA Component Model in der bereits bekannten Form. Dabei wird auf die Darstellung der Unterstützung einfacher Vererbung zwischen Component Types verzichtet.

Aufbauend auf diesen strukturellen Grundlagen differenziert CCM ähnlich wie EJB zwischen Kategorien von Komponenten, die sich hinsichtlich ihrer Aufgaben und ihres Lebenszyklus unterscheiden. *Service Components* haben vergleichbar mit Stateless Session Beans keinen konversationalen Zustand und werden pro eingehendem Aufruf instanziiert. *Session Components* unterstützen die Verarbeitung längerer Interaktionsfolgen durch einen (transienten) Zustand und entsprechen daher Stateful Session Beans. Des Weiteren betrachtet CCM neben den bereits aus EJB als Entity Beans bekannten *Entity Components* noch *Process Components*, die ähnlich wie Entity Components über einen per-

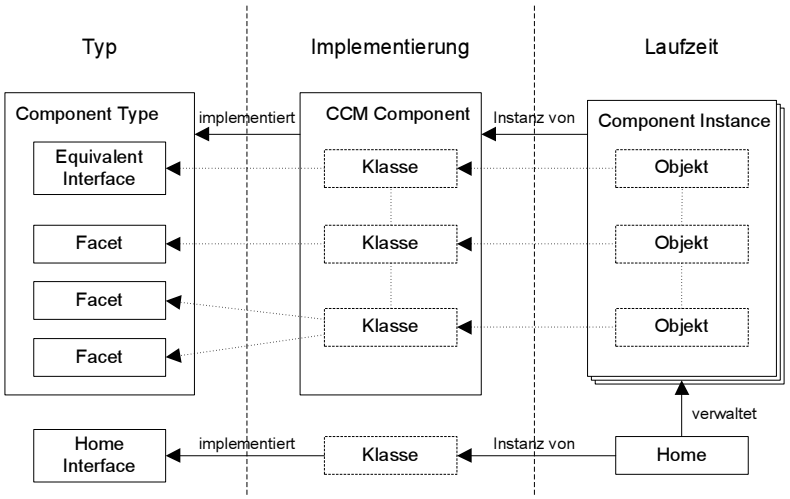


Abbildung 3.8: Komponentenmodell von CORBA Component Model (CCM)

sistenten Zustand verfügen, jedoch ihre Identität nicht in Form eines Fremdschlüssels veröffentlichen. Die Lebensdauer einer Process Component entspricht der Lebensdauer des unterstützten (Geschäfts-)Prozesses.

Web Services Zunehmend werden *Web Services* [GSB+02] als neuer Erscheinungstyp von Komponenten diskutiert. Ein Web Service ist ein Softwaresystem, dessen öffentliche Schnittstellen auf der Grundlage von XML beschrieben sind. Andere Softwaresysteme können Web Services anhand ihrer Beschreibung finden und mit ihnen durch den Austausch von XML-Nachrichten, die mittels Internetprotokollen übertragen werden, interagieren [W3C03]. Die Schnittstelle eines Web Services ist dabei durch einen oder mehrere *Port Types* definiert, die – ähnlich wie Schnittstellen in der objektorientierten Programmierung – die Operationen des Web Services spezifizieren. Web Services definieren damit eine Abstraktionsschicht, die Details der Implementierungs- und Einsatzplattform des bei Nutzung des Web Services ausgeführten Anwendungscodes vor Clients verbirgt.

Der Anwendungsentwicklung auf der Grundlage von Web Services liegt die so genannte *Web Service Architecture* [W3C03] zugrunde, die sich aus drei miteinander interagierenden Parteien zusammensetzt. Der *Anbieter* eines Web Services veröffentlicht eine Beschreibung des bereitgestellten Dienstes in einer *Service Registry*, die von *Nutzern* nach geeigneten Web Services durchsucht wird. Der Nutzer bindet einen gefundenen Web Service schließlich (u. U. dynamisch) in seine Anwendung ein.

Die Tatsache, dass Web Services ihre Dienste über standardisierte Schnittstellen veröffentlichen, legt eine Einordnung von Web Services als neue Form von (zustandslosen) Prozesskomponenten nahe, die ihre Dienste insbesondere über das Internet bereitstellen. Szyperski [Szy02] hebt allerdings hervor, dass Web Services keine Abhängigkeiten von anderen Web Services spezifizieren und diesbezüglich in höherem Maße abgeschlossen sind als „traditionelle“ Komponenten (vgl. Definition 3.1). Als Konsequenz dieser Abgeschlossenheit lassen sich Web Services nur in der angebotenen Form nutzen, ohne kontextspe-

zifische Anpassungen vornehmen zu können. Eine Rekonfiguration von Web Services, die ihre Wiederverwendung in einem anderen Kontext ermöglichen könnte, würde die explizite Veröffentlichung von Kontextabhängigkeiten sowie die Möglichkeit der Verknüpfung erwarteter Schnittstellen des Web Services an angebotene Schnittstellen anderer Web Services erfordern.

Da Web Services auf der Grundlage dieser Überlegungen weniger als Komponenten, sondern vielmehr als Schnittstellentechnologie einzuordnen sind, verzichten wir im Rahmen dieser Arbeit auf ihre weitere Berücksichtigung. Es soll dennoch angemerkt werden, dass die Suche nach Web Services in der Service Registry aufgrund der Mängel der Beschreibungssprache *Web Service Description Language (WSDL)* sowie des Suchdienstes *Universal Description, Discovery, and Integration (UDDI)* ein aktuelles Forschungsgebiet darstellt (vgl. z. B. [HYP01]). Mit Sprachen wie *Web Service Choreography Interface (WSCI)* [W3C02] und *Business Process Execution Language for Web Services (BPEL4WS)* [ACD+03] existieren jedoch interessante Vorschläge zur Beschreibung der Kombination von Web Services für die Abwicklung komplexer Geschäftsprozesse.

3.4 Komponentenbeschreibungssprachen

Die Black-Box-Wiederverwendbarkeit einer Komponente setzt die Verfügbarkeit einer angemessenen Beschreibung ihrer Struktur und ihres Verhaltens auf der syntaktischen (technischen) und semantischen (fachlichen) Ebene voraus. Was dabei der Begriff „angemessen“ meint, hängt von den mit der eingesetzten Beschreibungssprache verfolgten Zielen ab und variiert infolgedessen von Ansatz zu Ansatz. So sind die Beschreibungssprachen aktueller Komponententechnologien primär auf die Unterstützung des Deployments von Komponenten ausgerichtet. Auf Forschungsseite lassen sich formal geprägte Ansätze von eher fachlich orientierten Ansätzen unterscheiden. Während erstere die Interoperabilität von Komponenten in den Vordergrund stellen und bzgl. ihrer Komposition an Beweisen von Eigenschaften wie Lebendigkeit (Liveness) und Abwesenheit von Verklemmungen (Deadlocks) interessiert sind, betonen letztere fachliche und wirtschaftliche Aspekte.

In diesem Abschnitt skizzieren wir ausgehend von einer kurzen Charakterisierung der Beschreibungssprachen aktueller Komponententechnologien zunächst alternative Ansätze zur Verhaltenbeschreibung, denen wir entsprechende Beschreibungssprachen aus der Forschung zuordnen. Anschließend gehen wir näher auf einen Vorschlag des Arbeitskreises 5.10.3 *Komponentenorientierte betriebliche Anwendungssysteme* der *Gesellschaft für Informatik (GI)* zur umfassenden Spezifikation von Fachkomponenten ein. Schließlich bewerten wir die vorgestellten Ansätze zur Beschreibung von Komponenten hinsichtlich ihrer Eignung für die Komponentensuche.

3.4.1 Beschreibungssprachen aktueller Komponententechnologien

Der Fokus der Beschreibungssprachen aktueller Komponententechnologien liegt auf der Unterstützung des Deployments von Komponenten durch die Spezifikation von strukturellen Eigenschaften wie Schnittstellen, Kontextabhängigkeiten oder Konfigurationen. Fachliche oder wirtschaftliche Aspekte des Einsatzes von Komponenten, die z. B. für die Komponentensuche relevant sind, werden dagegen nicht betrachtet.

Die Komponententechnologie COM spezifiziert mit *COM IDL* eine Beschreibungssprache, die eine Erweiterung der durch die *Open Software Foundation (OSF)* im Kontext der *Distributed Computing Environment (DCE)* definierten *IDL (Interface Definition Language)* darstellt [Szy02]. COM IDL kann zum einen als Grundlage der Erzeugung von Typbibliotheken dienen, die zur Übersetzungs- und Laufzeit Typinformationen über Schnittstellen und Klassen bereitstellen, und zum anderen zur Generierung von Stubs und Proxies eingesetzt werden, die bei (über Prozess- oder Rechnergrenzen) verteilten Anwendungen zum Einsatz kommen. In .NET wird COM IDL durch die direkte Beschreibung von verwalteten Modulen mit Metadaten abgelöst. Diese Metadaten umfassen die durch ein Modul exportierten Typen, deren Methoden, Felder und Eigenschaften sowie Verweise auf externe Typen mitsamt der sie definierenden Assemblies. Da .NET den Anspruch erhebt, dass jedes verwaltete Modul selbstbeschreibend sein muss, ist die Beschreibung verwalteter Module durch Metadaten im Gegensatz zur Verwendung von COM IDL zur Erzeugung von COM-Typbibliotheken nicht optional [Pro02].

Die EJB-Architektur unterscheidet sechs verschiedene Rollen, die am Entwicklungs- und Einsatzprozess einer komponentenbasierten Anwendung beteiligt sind. Akteure, die diese Rollen einnehmen, kommunizieren durch den Austausch so genannter *Deployment Descriptors*, die entsprechend dem Anwendungsentwicklungs- und -einsatzprozess fortgeschrieben werden. Deployment Descriptors sind XML-Dokumente, die strukturelle Informationen zur Beschreibung einzelner Enterprise Beans und ihrer kompositionellen Verknüpfung beinhalten können. Zu diesen Informationen gehören die Nennung der in einer Installationseinheit enthaltenen Beans einschließlich der von ihnen unterstützten Home und Component Interfaces sowie die Spezifikation der Relationships zwischen Entity Beans und der Abhängigkeiten enthaltener Beans von externen Beans. Darüber hinaus können in einem Deployment Descriptor ein Sicherheitskonzept, das einzelnen Benutzerrollen spezifische Berechtigungen für den Zugriff auf Methoden einräumt, sowie das Transaktionsverhalten von Methoden definiert werden.

Das CCM sieht verschiedene XML-Dokumente vor, auf deren Grundlage einzelne Komponenten beschrieben sowie komponentenbasierte Anwendungen komponiert und konfiguriert werden. So spezifizieren *Component Descriptors* die Charakteristika einer Komponente zur Konstruktions- und Einsatzzeit. Dabei werden strukturelle Eigenschaften einzelner Komponenten durch die Spezifikation ihrer Ports und evtl. vorhandener Vererbungsbeziehungen zu anderen Komponenten beschrieben. Zur Unterstützung ihres Einsatzes in einem Container können ergänzend z. B. Informationen zum Transaktionsverhalten einer Komponente angegeben werden. Mit Hilfe eines *Component Assembly Descriptors* werden Anwendungen durch die Angabe genutzter Komponenten sowie die Verknüpfung von Instanzen oder Homes dieser Komponenten definiert. *Property File Descriptors* spezifizieren Werte von Attributen, die zur Konfiguration von Komponenten oder deren Homes eingesetzt werden.

3.4.2 Exkurs: Alternative Ansätze zur Verhaltensbeschreibung

Da die Beschreibung struktureller Eigenschaften von Komponenten aus Sicht der Forschung nur von beschränktem Interesse ist, konzentrieren sich entsprechende Arbeiten vielfach auf die Spezifikation des Verhaltens von Komponenten und deren Komposition.

Dabei können mindestens vier alternative Ansätze unterschieden werden, die wir im Folgenden gemeinsam mit entsprechenden Sprachen vorstellen wollen:

Zusicherungen HOARE schlägt in seiner Arbeit zum Beweis formaler Eigenschaften von Software [Hoa69] vor, die Semantik von Operationen in Form logischer Zusicherungen über dem Zustandsraum der Programmausführung zu spezifizieren (*axiomatische Semantik*). Dabei kann eine *Korrektheitsformel* (*Hoaresches Tripel*)

$$\{P\}\ op\ \{Q\}$$

wie folgt interpretiert werden: „Wenn die Operation *op* in einem Zustand gestartet wird, der die Zusicherung P erfüllt, dann ist bei Beendigung der Operation ein Zustand erreicht, in dem die Zusicherung Q gilt.“ Die Zusicherungen P bzw. Q werden auch als Vor- und Nachbedingung der Operation *op* bezeichnet.

Dieser zustandsorientierte Ansatz bildet die Grundlage für eine Vielzahl von Arbeiten zur Beschreibung des Verhaltens von Software, z. B. auf dem Gebiet der Programmverifikation und der formal gestützten (objektorientierten) Softwareentwicklung. So sind Korrektheitsformeln zentraler Bestandteil von Beweisskizzen in der Programmverifikation [AO94] sowie der Softwareentwicklungsmethode des *Design by Contract* [Mey92], bei der Softwareelemente als Anbieter und Nutzer von Diensten auf der Grundlage formaler Verträge miteinander kooperieren. Neben Vor- und Nachbedingungen berücksichtigt das *Design by Contract* auch Klasseninvarianten, die den unveränderlichen Zustand einer Klasse bzw. ihrer Instanzen beschreiben. Die objektorientierte Programmiersprache *Eiffel* [Mey91] folgt diesem Gedanken des *Design by Contract* und stellt entsprechende syntaktische Konstrukte für die Formulierung von Vor- und Nachbedingungen sowie Klasseninvarianten zur Verfügung. Einen ähnlichen Ansatz verfolgt die *Java Modeling Language (JML)* [LBR99], mit der sich das Verhalten von Java-Schnittstellen durch deren Annotation mit Zusicherungen spezifizieren lässt. Die *Object Constraint Language (OCL)* [OMG03] gestattet als Teilsprache der UML ebenfalls die formale Spezifikation der Semantik von Klassen und deren Methoden durch Vor- und Nachbedingungen sowie Klasseninvarianten. Entsprechende OCL-Zusicherungen werden mittels vordefinierter Prädikate über den Attributen, Assoziationen und Methoden einer Klasse formuliert. Als Beispiel für den Einsatz von OCL zur Verhaltensbeschreibung möge die folgende Spezifikation eines unbegrenzten Stacks mit zwei Methoden `push()` und `pop()` dienen:

```
context Stack::push(o: Object): void
  post:
    elements = elements@pre->append(o)

context Stack::pop(): Object
  pre:
    elements->notEmpty()
  post:
    result = elements@pre->last()
      and elements = elements@pre->subSequence
                        (1, elements@pre->size() - 1)
```

Die Nachbedingung der Methode `push()` sichert zu, dass das neue Objekt `o` den Elementen des Stacks hinzugefügt wird. Die Methode `pop()` verlangt in ihrer Vorbedingung, dass die Menge der Objekte auf dem Stack nicht leer ist, und sichert dann zu, das zuletzt auf den Stack gelegte Objekt zurückzuliefern und aus der Menge der Objekte auf dem Stack zu entfernen.

Endliche Automaten Die Spezifikation des Verhaltens von Operationen mittels logischer Zusicherungen stellt einen ausdrucksstarken Ansatz auf einer soliden formalen Grundlage dar. Allerdings sind die resultierenden Verhaltensbeschreibungen nicht operationell, d. h. an einem abstrakten Maschinenmodell orientiert (*operationelle Semantik*), und deshalb für den menschlichen Betrachter oft nur schwer nachvollziehbar. Eine Alternative zu dieser zustandsorientierten Beschreibung des Verhaltens von Software ist die verhaltensorientierte Spezifikation der Ausführbarkeit von Operationen in Form *endlicher Automaten*. Ein endlicher Automat definiert eine endliche Menge von Zuständen, die durch Zustandsübergänge (Transitionen) miteinander verknüpft sind. Ein Zustandswechsel wird durch das „Schalten“ einer Transition aufgrund der Ausführung der ihr zugeordneten Aktion (hier: Operation) ausgelöst. Der Fokus dieser Beschreibungsform von Software liegt dabei weniger auf den Zuständen, die im Allgemeinen nur mit abstrakten Namen bezeichnet werden, sondern vielmehr auf den zulässigen Operationssequenzen. Nierstrasz nutzt in [Nie95] endliche Automaten, um die Verfügbarkeit der Dienste von Objekten durch reguläre Prozesse, d. h. durch Prozesse mit einer endlichen Anzahl von Zuständen zu beschreiben. Auf der Grundlage der durch diese Beschreibungen spezifizierten regulären Typen definiert er eine Subtyping-Relation, die die Substituierbarkeit von Objekten formalisiert.

Harel hat mit *Statecharts* [HG97] ein ausdrucksstarkes Automatenmodell zur Spezifikation der Dynamik objektorientierter Softwaresysteme entwickelt, das als *Zustandsmaschinen* (*State Machines*) auch Eingang in die UML gefunden hat. Statecharts verfügen über einen mächtigeren Transitionsbegriff als klassische endliche Automaten, der sich im Sinne einer *ECA-Regel* (Event, Condition, Action) aus einer Ereignisdefinition (Event, Trigger), einem Wächter (Guard Condition) und einer bei Eintreten des Ereignisses und Erfüllung des Wächters (neben dem Zustandswechsel) ausgeführten Aktionsbeschreibung (Action Sequence) zusammensetzt. Im Gegensatz zu den Zuständen eines klassischen Automaten können Zustände eines Statecharts zudem mittels sequentieller und paralleler Dekomposition auch hierarchisch zerlegt werden. Ein Zustand wird dann durch ein oder mehrere parallele Statecharts verfeinert.

Abbildung 3.9 stellt die verhaltensorientierte Spezifikation des unbegrenzten Stacks als Statechart dar. Zur Abbildung des unendlichen Zustandsraums eines unbegrenzten Stacks durch ein endliches Automatenmodell behelfen wir uns dabei mit der Einführung einer Zählervariablen `size`, die die Anzahl der aktuell auf dem Stack befindlichen Objekte vorhält. Im Zustand `leer` steht nur die Methode `push()` zur Verfügung. Ihr Aufruf führt neben dem Wechsel in den Zustand `nicht leer`, in dem sowohl die Methode `push()` als auch die Methode `pop()` zur Ausführung bereit sind, zur Inkrementierung der Zählervariablen `size`. Ein Aufruf der Methode `pop()` dekrementiert die Zählervariable und führt abhängig von der Anzahl der aktuell auf dem Stack gespeicherten Elemente zur Rückkehr in den Zustand `leer`.

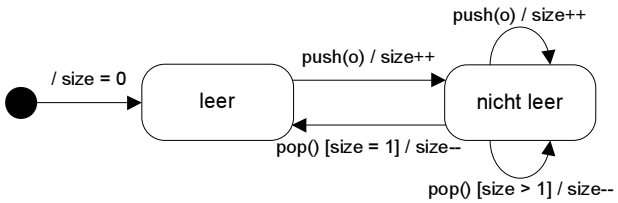


Abbildung 3.9: Spezifikation eines unbegrenzten Stacks als Statechart

Als weiteres Automatenmodell seien hier noch die ebenfalls ausdrucksmächtigen, jedoch weniger intuitiv verständlichen *Petri-Netze* [Rei85] erwähnt. Petri-Netze nutzen einen verteilten Zustands- und Transitionsbegriff, bei dem eine globale Transition zwischen globalen Zuständen ausgeführt wird, indem eine Menge gleichzeitig aktivierter lokaler Transitionen zwischen lokalen Teilzuständen parallel „schalten“ [Old91].

Prozessterme Eine weitere Alternative zur Spezifikation des Verhaltens von Software besteht darin, die autonomen, aktiven Einheiten eines Softwaresystems als nebenläufige Prozesse aufzufassen, deren Verhalten durch so genannte *Prozessterme* spezifiziert wird. Prozessterme basieren auf einer *Prozessalgebra* [BW90] wie z. B. *CSP (Communicating Sequential Processes)* [Hoa85], *CCS (Calculus of Communicating Systems)* [Mil89] oder dem *π-Kalkül* [MPW92] und gestatten ähnlich wie Automatenmodelle eine operationelle Darstellung des Verhaltens (*operationelle Semantik*). Eine Prozessalgebra ist dabei eine formalisierte mathematische Sprache, die zum einen die Formulierung von Prozessstermen durch die Bereitstellung von Konstanten sowie Operatoren für die Komposition komplexerer Terme aus einfacheren Termen und zum anderen deren Vergleich durch die Definition von Axiomen unterstützt. Der unbegrenzte Stack lässt sich mittels CSP durch $STACK = P_{\langle\rangle}$ spezifizieren (vgl. [Hoa85]), wobei gilt:

$$P_{\langle\rangle} = push?x \longrightarrow P_{\langle x\rangle}$$
$$P_{\langle x\rangle^\frown s} = (\, pop!x \longrightarrow P_s \mid push?y \longrightarrow P_{\langle y\rangle^\frown\langle x\rangle^\frown s}\,)$$

Dabei beschreiben die Terme P_{state} das ausführbare Verhalten des Stacks in den jeweiligen, durch die aktuelle Belegung determinierten Zuständen. Der (initial) leere Stack ist zunächst nur bereit, die (Eingabe-)Aktion $push?x$ auszuführen, mit der ein neues Element x auf den Stack gelegt wird. Enthält der Stack ein oberstes Element x (und eine nicht näher bestimmte Sequenz s weiterer Elemente), so kann alternativ das Element x durch Ausführung der (Ausgabe-)Aktion $pop!x$ entnommen oder aber ein weiteres Element y mittels $push?y$ hinzugefügt werden.

Canal et al. [CFTV00] setzen Prozessterme in ihrer Arbeit zur Erweiterung der Schnittstellen von CORBA-Objekten um Protokollaspekte ein. Sie spezifizieren die Reihenfolge der von einem Objekt erwarteten bzw. ausgelösten Methodenaufrufe mittels einer syntaktisch angereicherten Version des polyadischen π-Kalküls, bei dem unabhängige Prozesse über Kommunikationskanäle miteinander kommunizieren können.

Der Fokus von *Architekturbeschreibungssprachen (Architecture Description Languages, ADLs)* [SG96] liegt weniger auf der detaillierten Beschreibung des Verhaltens ein-

zelner Komponenten, sondern vielmehr auf der abstrakten Beschreibung komplexer Softwaresysteme durch die Darstellung ihres Aufbaus aus einzelnen, miteinander interagierenden Komponenten. Eine Architekturbeschreibungssprache, die neben der Struktur eines Softwaresystems insbesondere die Interaktionen zwischen den beteiligten Komponenten spezifiziert, ist *Darwin* [MDEK95]. Basiselemente einer Architektur in Darwin sind Komponenten, die angebotene Dienste hinter einer wohldefinierten (direkten) Schnittstelle (*provided interface*) verbergen und benötigte Dienste explizit spezifizieren (*required interface*). Das Verhalten einer einfachen Komponente ergibt sich aus Beschreibungen der angebotenen und benötigten Dienste und der Spezifikation der Bereitstellung bzw. Anforderung dieser Dienste im Sinne eines Protokolls. Komplexe Komponenten lassen sich konstruieren, indem angebotene Dienste einer Komponente an benötigte Dienste einer anderen Komponente gebunden werden. Verhaltensbeschreibungen werden in Darwin nicht direkt mit einer Prozessalgebra formuliert, jedoch definiert Darwin eine Übersetzersemantik in den π-Kalkül. Hinsichtlich der Spezifikation des Verhaltens von Komponenten stark von Darwin beeinflusst ist die Komponentenbeschreibungssprache *Simple Component Description Language (SCDL)* [Rit00]. Grundlage der Verhaltensbeschreibung mit SCDL ist der π-Kalkül bzw. die darauf aufsetzende Programmiersprache *Pict*.

Die Architekturbeschreibungssprache *Wright* [AG97] baut direkt auf dem Component/Connector-Komponentenmodell (vgl. Abschnitt 3.3.1) auf. Eine Wright-Spezifikation beschreibt die Schnittstelle einer Komponente als Menge von Interaktionspunkten (Ports), deren Verhalten durch ein Protokoll auf Basis einer Teilmenge von CSP definiert ist. Darüber hinaus lässt sich optional spezifizieren, wie die Interaktionen auf diesen Interaktionspunkten durch die Komponente zu komplexeren Berechnungen kombiniert werden. Konnektoren werden vergleichbar mit einem Framework durch eine Menge von Rollen und eine so genannte „Glue"-Spezifikation spezifiziert, wobei die Rollen das erwartete lokale Verhalten der interagierenden Komponenten beschreiben und die Glue-Spezifikation als zusammenhaltende Kraft („Leim") die Aktivitäten der einzelnen Rollen koordiniert und damit den Kontrollfluss definiert. Dabei besteht die Glue-Spezifikation eines Konnektors aus zwei Teilen: einem endlichen Interaktionsprotokoll auf Basis von CSP sowie einem Prädikat über die durch dieses Protokoll generierten Traces (Trace-Spezifikation).

Trace-Spezifikationen Der mit *Trace-Spezifikationen* [Old91] verfolgte Ansatz zur Beschreibung von Verhalten geht von der Beobachtung aus, dass unterschiedliche Prozessterme dieselbe Menge von Kommunikationssequenzen (*Traces*) bezeichnen können. Trace-Spezifikationen abstrahieren daher von einzelnen Prozesstermen und beschreiben Verhalten mittels Prädikaten einer Trace-Logik als Mengen erzeugbarer Traces. Eine Trace-Spezifikation für einen unbegrenzten Stack mit den Aktionen (Methoden) *push* und *pop* könnte wie folgt aussehen:

$$S := |\,(\text{pref}\,h) \downarrow \{pop\}\,| \leq |\,(\text{pref}\,h) \downarrow \{push\}\,|$$

Diese Trace-Spezifikation referenziert die erzeugten Traces über die Variable h (für „history") und verlangt, dass die Anzahl von *pop*-Aktionen in jedem Präfix von h kleiner oder gleich der Anzahl von *push*-Aktionen ist. Damit sind z. B. *push.push.pop* und *push.pop.push.push* gültige Traces, während *push.pop.pop* und *pop.push* die Spezifikation

verletzen. Als Nachteil von Trace-Spezifikationen ist festzustellen, dass sie die in Prozesstermen gewissermaßen als Produktionsvorschrift erfasste operationelle Beschreibung der schrittweisen Erzeugung von Traces einbüßen.

Krämer [Krä98] nutzt Trace-Spezifikationen, um die Protokolle von CORBA-Schnittstellen durch so genannte *Synchronisation Constraints* zu beschreiben. Wie bereits erwähnt werden Trace-Spezifikationen zudem in der Architekturbeschreibungssprache Wright zur Beschreibung des Verhaltens von Konnektoren eingesetzt.

3.4.3 Memorandum zur vereinheitlichten Spezifikation von Fachkomponenten

Während sich die bislang genannten Ansätze zur Beschreibung von Komponenten auf das Deployment bzw. die Spezifikation des Verhaltens beschränken, schlägt das Memorandum zur *Vereinheitlichten Spezifikation von Fachkomponenten* [ABC+02] des GI-Arbeitskreises 5.10.3 *Komponentenorientierte betriebliche Anwendungssysteme* einen umfassenden Rahmen für die Beschreibung von Fachkomponenten vor. Dieser Spezifikationsrahmen sieht sieben Beschreibungsebenen vor, auf denen die unterschiedlichen Aspekte einer Fachkomponente dokumentiert werden (vgl. Abbildung 3.10).

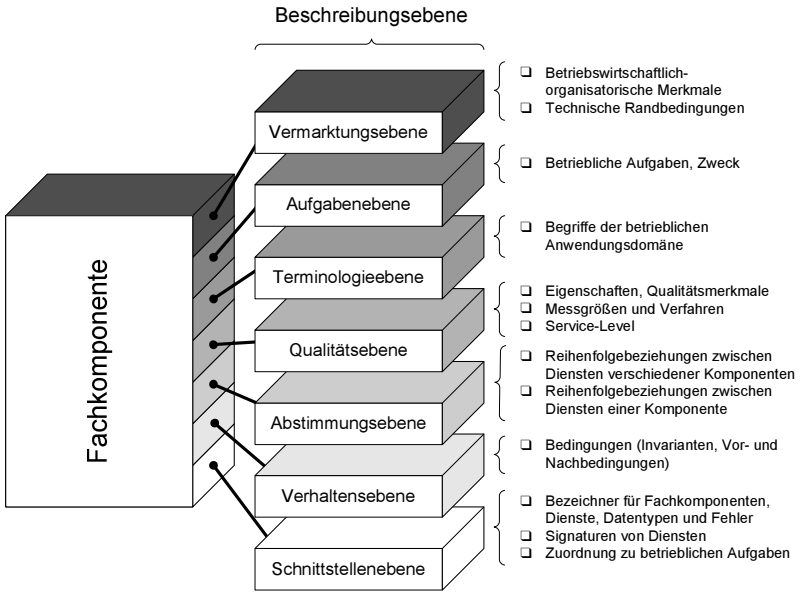


Abbildung 3.10: Ebenen der Spezifikation von Fachkomponenten (nach [ABC+02])

Im Folgenden stellen wir übersichtsartig Gegenstand und vorgeschlagene Notation der einzelnen Beschreibungsebenen vor:

- *Schnittstellenebene*: Gegenstand der Schnittstellenebene ist die syntaktische Beschreibung der Schnittstelle einer Komponente durch die Vereinbarung von Signaturen der angebotenen Dienste, verwendeten Datentypen und Ausnahmesituatio-

nen (Exceptions). Als Notation auf der Schnittstellenebene wird die Verwendung der *OMG Interface Definition Language (IDL)* [OMG02a] vorgeschlagen.

- *Verhaltensebene*: Vereinbarungen auf der Verhaltensebene dienen der näheren Beschreibung des Verhaltens einer Komponente durch Invarianten sowie Vor- und Nachbedingungen. Für die Spezifikation des Verhaltens wird die Nutzung der OCL vorgeschlagen, wobei zur Erhöhung der Akzeptanz gefordert wird, dass alle OCL-Ausdrücke ergänzend in natürlicher Sprache formuliert werden.

- *Abstimmungsebene*: Auf der Abstimmungsebene werden die Reihenfolge der Verwendung von Diensten und die Synchronisationserfordernisse zwischen Diensten geregelt. Zur Notation von Sachverhalten auf der Abstimmungsebene wird die Verwendung einer um temporale Operatoren erweiterten OCL [CT01] vorgeschlagen.

- *Qualitätsebene*: Nicht-funktionale Eigenschaften einer Fachkomponente wie z. B. Verfügbarkeit, Wartbarkeit oder Performance werden auf der Qualitätsebene dokumentiert. Ein konkreter Notationsvorschlag wird aufgrund der Vielfalt der zu spezifizierenden Sachverhalte nicht gegeben.

- *Terminologieebene*: Die Terminologieebene dient als zentrales Verzeichnis der auf den anderen Beschreibungsebenen verwendeten Fachbegriffe. Als Notation auf der Terminologieebene wird die Verwendung einer Fachnormsprache [Ort97] empfohlen, die u. a. ein Lexikon inhaltlich geklärter Fachbegriffe definiert.

- *Aufgabenebene*: Gegenstand der Aufgabenebene ist die Dokumentation der von einer Fachkomponente unterstützten betrieblichen Aufgaben sowie deren Zuordnung zu den auf der Schnittstellenebene spezifizierten Diensten. Für die Spezifikation auf der Aufgabenebene wird wieder der Einsatz einer Fachnormsprache vorgeschlagen. Betriebliche Aufgaben sollen auf der Grundlage ihrer rationalen Grammatik und der auf der Terminologieebene definierten Fachbegriffe beschrieben werden.

- *Vermarktungsebene*: Auf der Vermarktungsebene werden alle Merkmale einer Fachkomponente spezifiziert, die aus betriebswirtschaftlich-organisatorischer Sicht erforderlich sind. Dazu gehören die Anwendungsdomäne, Systemanforderungen und Vertragskonditionen. Als Notation ist eine einfache Tabellenform vorgesehen.

Das Fachkomponenten-Memorandum steckt in seiner gegenwärtigen Fassung lediglich einen recht groben Rahmen für die Spezifikation von Fachkomponenten ab, der noch weiterer Konkretisierung bedarf.

3.4.4 Kritische Betrachtung

Die vorgestellten Ansätze zur Beschreibung von Komponenten werden der Anforderung, die Spezifikation der Schnittstellen und Kontextabhängigkeiten einer Komponente auf syntaktischer (technischer) und semantischer (fachlicher) Ebene zu ermöglichen, aus Sicht der Komponentensuche nur in unzureichendem Maße gerecht. So konzentrieren sich COM IDL und die verschiedenen XML-Formate von EJB und CCM ausschließlich auf die für

das Deployment von Komponenten relevanten syntaktischen Aspekte und lassen die Repräsentation semantischer Eigenschaften wie das Verhalten oder den fachlichen Bezug einer Komponente vollständig außer Acht. Die vorgestellten Forschungsarbeiten zur Repräsentation des Verhaltens schlagen verschiedene Ansätze zur Darstellung der formalen Semantik von Software vor, betrachten aber ebenso wenig die fachliche Semantik, die den Bezug einer Komponente zu einem Anwendungsgebiet definiert. Bezüglich der Repräsentation der formalen Semantik ist ergänzend anzumerken, dass Zusicherungen und Trace-Spezifikationen als nicht operationelle Darstellungsformen bei der Spezifikation von realen Komponenten schnell zu komplex für einen menschlichen Betrachter werden (vgl. hierzu auch [Ack01]). Das Memorandum zur vereinheitlichten Spezifikation von Fachkomponenten stellt als einzige der vorgestellten Beschreibungssprachen einen Bezug zwischen Technologie und Anwendungsdomäne her, leidet aber unter den bereits genannten Nachteilen der Verhaltensbeschreibung mittels Zusicherungen.

Aus Sicht der Komponentensuche erscheinen einige Aspekte der untersuchten Sprachen dennoch für die Auswahl bzw. den Entwurf einer Komponentenbeschreibungssprache relevant. So stellt die im Fachkomponenten-Memorandum aufgegriffene Idee, die fachliche Bedeutung von Software durch den Einsatz einer Normsprache wiederzugeben, einen durchaus interessanten Ansatz dar. Die operationelle Beschreibung des Verhaltens von Klassen und Schnittstellen auf der Grundlage von Prozesstermen oder Statecharts bietet insbesondere durch die Möglichkeit zur graphischen Veranschaulichung den Vorteil, auch für einen menschlichen Betrachter verständlich zu sein. Schließlich ist die von einigen Architekturbeschreibungssprachen erzielte Integration struktureller und verhaltensmäßiger Aspekte von (komplexen) Komponenten auch für eine Komponentenbeschreibungssprache erstrebenswert.

3.5 Zusammenfassung

In diesem Kapitel haben wir die für diese Arbeit wesentlichen Grundlagen der Komponentensoftware dargestellt. Dabei haben wir – aufbauend auf einer Einführung der zentralen Grundbegriffe – zunächst den Makroprozess der komponentenbasierten Softwareentwicklung betrachtet, der insbesondere durch den Gedanken geprägt ist, Komponenten auf elektronischen Marktplätzen zu handeln. Mit Blick auf die Anforderungen an eine möglichst allgemeine Komponentenbeschreibungssprache, die die umfassende Spezifikation von Komponenten auf Komponentenmärkten gestattet, haben wir verschiedene Komponentenmodelle hinsichtlich ihrer strukturellen Merkmale analysiert. Gegenstand dieser Untersuchung waren einerseits konzeptionelle Komponentenmodelle aus der Forschung und andererseits die Komponentenmodelle der Komponententechnologien COM, .NET, EJB und CCM. Schließlich haben wir existierende Ansätze zur Beschreibung von Komponenten aus Praxis und Forschung betrachtet und hinsichtlich ihrer Eignung für die Komponentensuche bewertet. Dabei hat sich insbesondere gezeigt, dass aktuelle Komponentenbeschreibungssprachen die Anforderungen der Komponentensuche nur in unzureichendem Maße erfüllen.

Kapitel 4

Komponentensuche

Ziel dieses Kapitels ist eine Übersicht über den State of the Art in Praxis und Forschung auf dem Gebiet der Komponentensuche. Nach einer Einführung in die Komponentensuche in Abschnitt 4.1 skizzieren wir in Abschnitt 4.2 zunächst die Komponentensuche in der Praxis aktueller Komponentenmärkte. Abschnitt 4.3 gibt anschließend einen ausführlicheren Überblick über das Spektrum der Forschungsarbeiten zur Komponentensuche. Wir nehmen in Abschnitt 4.4 eine Bewertung der Suchverfahren aus Praxis und Forschung vor und beschließen das Kapitel in Abschnitt 4.5 mit einer Zusammenfassung.

4.1 Einführung

Die Suche nach wiederverwendbarer Software ist ein Forschungsgebiet, das seit mindestens zwei Jahrzehnten aktiv untersucht wird. Im Laufe dieser Zeitspanne haben der Einsatz neuer Entwicklungsparadigmen wie z. B. die objektorientierte Programmierung oder nun zunehmend die komponentenbasierte Softwareentwicklung, die Verfügbarkeit neuer Technologien wie das Internet oder einfach nur die Steigerung der Leistungsfähigkeit verfügbarer Hardware neue Herausforderungen und Rahmenbedingungen für die Forschung auf diesem Gebiet definiert. So haben objektorientierte und komponentenbasierte Softwareentwicklung die Organisation von Bibliotheken wiederverwendbarer Software maßgeblich beeinflusst, während das Internet neue Wege für die Distribution von Software ermöglicht hat. Da wir uns im Rahmen dieser Arbeit mit der Suche nach Komponentensoftware beschäftigen, sprechen wir im Folgenden von Komponentensuche anstelle der allgemeineren Suche nach wiederverwendbarer Software und von Komponenten-Repositories statt von Bibliotheken wiederverwendbarer Software. Die Mehrzahl der in diesem Kapitel gemachten Ausführungen gilt dennoch auch für die allgemeine Suche nach Software.

Das zentrale Prinzip jeglichen Ansatzes zur Komponentensuche ist der Vergleich einer Anfragespezifikation, in der ein (Wieder-)Verwender ein vorliegendes Problem formuliert, mit einer Repräsentation einer potentiellen Softwarelösung, die durch den Betreiber eines Komponenten-Repositories oder den Komponentenhersteller selbst angefertigt wird. Aufgrund von Effizienzüberlegungen wird dabei im Allgemeinen nicht die Komponente selbst in den Vergleich einbezogen, sondern lediglich eine abstrakte Komponentenbeschreibung, die die wesentlichen Eigenschaften der Software zusammenfasst. Henninger [Hen94]

identifiziert als grundlegendes Problem der Komponentensuche die konzeptuelle Lücke zwischen dem *Situationsmodell*, das die Sicht des Verwenders auf seine Anforderungen wiedergibt und in eine Anfragespezifikation überführt werden muss, und dem *Systemmodell*, das die Perspektive des Komponentenherstellers auf eine von ihm entwickelte Komponente beschreibt und in eine Komponentenbeschreibung umformuliert werden muss. Von zentraler Bedeutung für die Komponentensuche ist folglich die Reduzierung dieser konzeptuellen Lücke durch die Bereitstellung angemessener, d. h. dem Situations- bzw. Systemmodell angepasster Sprachen zur Formulierung von Anfragen bzw. zur Beschreibung von Komponenten sowie die Überbrückung der verbleibenden Lücke durch geeignete Suchverfahren. Dieser Zusammenhang wird in Abbildung 4.1 graphisch illustriert.

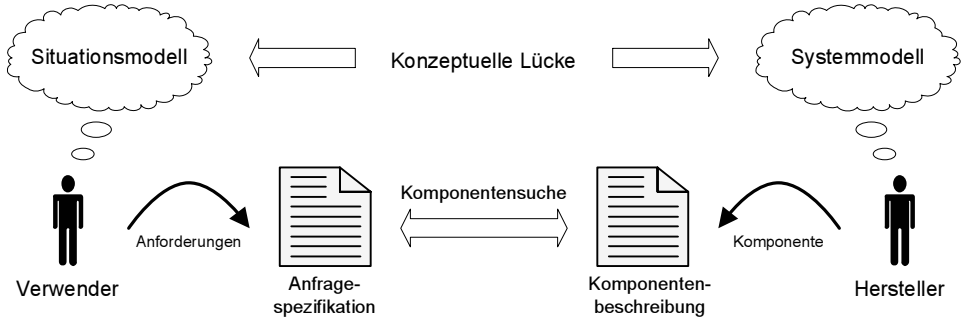


Abbildung 4.1: Problem der Komponentensuche

Die Formulierung von Anfragen, also die Spezifikation des Situationsmodells, lässt sich u. a. durch Angabe von

- Schlüsselwörtern,
- Attribut/Wert-Paaren,
- natürlichsprachlichen Aussagen bzw. Aussagemustern,
- Operationssignaturen,
- Ein-/Ausgabedaten,
- funktionalen Beschreibungen von Operationen,
- konzeptuellen Graphen und
- Entwürfen oder Programmmustern

erreichen (vgl. auch [MMM98]). Eng verknüpft mit der Wahl der Sprache zur Spezifikation von Suchanfragen ist der Charakter des für die Suche nach Komponenten eingesetzten Verfahrens. Um die Vielzahl untersuchter Suchverfahren einordnen zu können, identifizieren Mili et al. in [MMM98] sechs Kategorien:

- Verfahren des *Information Retrieval*: Übertragung traditioneller Methoden des Information Retrieval [SM83], die für die Suche nach Dokumenten wie z. B. Büchern eingesetzt werden, auf die Komponentensuche. Dabei werden Komponenten vereinfachend als spezielle Form von (Text-)Dokumenten betrachtet, ohne spezifische Eigenschaften wie z. B. ihre Ausführbarkeit oder Semantik bei der Suche zu berücksichtigen.

- *Deskriptive Verfahren*: Vergleich einer Anfrage auf Basis von Schlüsselwörtern mit Komponenten, die durch (strukturierte) Listen beschreibender Schlüsselwörter repräsentiert werden.

- Verfahren auf Grundlage der *operationellen Semantik*: Bewertung des Verhaltens einer Komponente bzgl. der durch eine Anfrage vorgegebenen Ein- und Ausgabedaten. Diese Verfahren basieren im Allgemeinen auf der Ausführbarkeit von Software, können sich aber auch auf abstrakten Maschinenmodellen abstützen.

- Verfahren auf Grundlage der *denotationellen Semantik*: Überprüfung einer semantischen Relation zwischen einer Anfrage und der abstrakten Beschreibung des Verhaltens einer Komponente. Diese Verfahren lassen sich auch auf nicht ausführbare Artefakte wie z. B. Spezifikationen anwenden.

- *Topologische Verfahren*: Identifikation von Komponenten, die ein Maß für den Abstand zu einer Anfrage im Vergleich mit anderen Komponenten minimieren. Die absolute Relevanz einer Komponente hängt damit nicht allein von der Anfrage, sondern auch von der relativen Relevanz anderer Komponenten ab.

- *Strukturelle Verfahren*: Auswahl einer Komponente aufgrund struktureller Ähnlichkeit mit der in einer Anfrage formulierten Vorstellung von einer geeigneten Softwarelösung.

Diese Kategorien von Verfahren für die Komponentensuche bestimmen bereits weitgehend den Charakter der für die Beschreibung von Komponenten und damit des Systemmodells eingesetzten Sprachen (vgl. auch Abschnitt 3.4).

Die Qualität eines Suchverfahrens wird – neben Komplexitätsbetrachtungen – auf der Grundlage zweier Kennzahlen bewertet, die ursprünglich aus dem Information Retrieval stammen. Die *Präzision (Precision)* eines Verfahrens ist durch den Anteil der relevanten gefundenen Komponenten an allen gefundenen Komponenten definiert, während die *Trefferquote (Recall)* den Anteil relevanter gefundener Komponenten an allen relevanten Komponenten angibt. Da mit der Erhöhung der Präzision im Allgemeinen die Trefferquote abnimmt (und umgekehrt), wird beim klassischen Information Retrieval versucht, einen möglichst guten Kompromiss zwischen diesen Kenngrößen zu erreichen. Bei der Suche nach Komponenten, die im Sinne einer Black-Box-Wiederverwendung genutzt werden sollen, ist man dagegen primär an einer hohen Präzision interessiert, um den für den Einsatz einer Komponente erforderlichen Anpassungsaufwand zu minimieren [BR89, GI94a, ZW97]. Da strukturelle Verfahren im Allgemeinen die Modifikation einer gefundenen Komponenten im Sinne der White-Box-Wiederverwendung voraussetzen, sind sie für die Anwendung auf unseren Komponentenbegriff (vgl. Definition 3.1) ungeeignet und werden daher im Folgenden nicht weiter betrachtet.

4.2 Komponentensuche in der Praxis aktueller Komponentenmärkte

Einhergehend mit dem zunehmenden Erfolg der komponentenbasierten Softwareentwicklung sind in den vergangenen Jahren eine Reihe internetbasierter Komponentenmärkte entstanden, von denen allerdings nur ein geringer Teil praktische Relevanz erreichen konnte (Übersichten über aktuelle Komponentenmärkte finden sich z. B. in [HM02] und [TUC03]). Obgleich wir uns in dieser Arbeit auf die Suche nach Fachkomponenten konzentrieren wollen, geben wir im Folgenden eine übersichtsartige Charakterisierung allgemeiner Komponentenmärkte hinsichtlich der Typen angebotener Komponenten, der für die Beschreibung dieser Komponenten verwendeten Sprachen sowie der eingesetzten Suchverfahren an. Für eine detaillierte Untersuchung aktueller Komponentenmärkte sei auf die Arbeit von Hau und Mertens [HM02] verwiesen.

Bezüglich der auf Komponentenmärkten wie z. B. ComponentSource [Com03], Sun iForce Partner Products Catalog [Sun03], SoftSelect [Sof03], SAP Software Partner Directory [SAP03], Software-Marktplatz [Nom03] oder dem Open-Source-Marktplatz SourceForge.net [Sou03] angebotenen Typen von Komponenten lässt sich eine große Diversifizierung feststellen. Das Spektrum angebotener Komponententypen umfasst mindestens:

- Fach- und Systemkomponenten auf Basis aktueller Komponententechnologien wie COM, .NET und EJB,
- GUI-Komponenten auf Grundlage der Technologien JavaBeans und ActiveX,
- Erweiterungsmodule für verschiedenartige Anwendungen wie z. B. ERP-Softwareprodukte wie z. B. SAP R/3 oder die integrierte Entwicklungsumgebung *Eclipse* sowie
- selbständig lauffähige Anwendungen, die nicht unter unseren Komponentenbegriff fallen.

Für die angebotenen Komponenten werden in der Regel Komponentenbeschreibungen angeboten, die sich an einem seitens des Marktplatzbetreibers vorgegebenen, im Allgemeinen jedoch sehr groben Beschreibungsschema orientieren. Diese Beschreibungen beinhalten im besten Fall Informationen zu:

- Produktbezeichnung,
- Hersteller,
- Funktionalität (Kurz- und Detailbeschreibung als Freitext),
- Kategorie/Branche/Anwendungsgebiet,
- Preis und Lizenzierung,
- Zertifizierung,
- Systemanforderungen,

- Kompatibilität, z. B. mit Betriebsystemen oder Application Servern,
- Reviews von Kunden sowie
- Evaluationsmöglichkeiten.

Da insbesondere die Kurz- und Detailbeschreibungen der Funktionalität von Komponenten als Freitext angegeben werden können, besteht die Möglichkeit (und gleichzeitig die „Gefahr"), das Schema diesbezüglich weitgehend frei zu interpretieren. Eine Einheitlichkeit der Beschreibungen, die den Vergleich von Komponenten ermöglichen würde, ist somit weder innerhalb eines Komponentenmarktes noch zwischen verschiedenen Marktplätzen gegeben. Zum Teil wird sogar gänzlich auf eine Komponentenbeschreibung verzichtet und lediglich auf den Hersteller verwiesen (siehe z. B. SoftSelect [Sof03]).

Mili et al. [MMM98] stellen fest, dass die Praxis der Komponentensuche durch die Nutzung einfacher „Ad-hoc-Suchverfahren" gekennzeichnet ist. Eine nähere Betrachtung aktueller Komponentenmärkte bestätigt diese Feststellung. Die Suche nach Komponenten wird dort allgemein nur durch einfache Verfahren wie

- Stichwortsuche in Freitextbeschreibungen,
- Klassifizierung mittels strukturierter Listen beschreibender Schlüsselwörter,
- Navigation durch Kategorien,
- Filtermechanismen sowie die
- Bewertung spezifischer Anforderungen in Kriterienkatalogen für Branchen, Fachgebiete und Funktionalität

unterstützt. Präzisere semantische Suchverfahren, wie sie ein Broker im Makroprozess komponentenbasierter Softwareentwicklung (vgl. Abschnitt 3.2) anbieten könnte, sind auf aktuellen Komponentenmärkten nicht anzutreffen [HM02]. Solche „intelligenten" Suchverfahren würden allerdings auch den Einsatz präziserer Komponentenbeschreibungen voraussetzen, die die durch eine Komponente realisierte Funktionalität möglichst „exakt" dokumentieren.

4.3 Komponentensuche in der Forschung

In diesem Abschnitt stellen wir eine repräsentative Auswahl verschiedener Ansätze zur Komponentensuche aus der Forschung vor. Mit Ausnahme der Kategorie der strukturellen Verfahren, die wir aufgrund ihrer Ausrichtung auf die White-Box-Wiederverwendung nicht betrachten, erläutern wir für jede der in Abschnitt 4.1 genannten Kategorien von Suchverfahren exemplarisch zumindest einen repräsentativen Ansatz. Für eine vollständigere Übersicht sei auf die Arbeit von Mili et al. [MMM98] verwiesen.

4.3.1 Faceted Classification

Ein grundlegendes Prinzip zur systematischen Organisation großer Mengen von Dingen ist das der Klassifizierung, wobei grundsätzlich zwei Arten von Klassifizierungsschemata unterschieden werden können. Bei der traditionellen Klassifizierung (*Enumerative Classification*) wird das Klassifizierungsschema durch Aufzählung und hierarchische Anordnung aller verfügbaren Klassen gewonnen, indem der betrachtete Weltausschnitt im Sinne eines Top-Down-Ansatzes in zunehmend feinere Klassen zerlegt wird. Bei der im Bibliothekswesen eingesetzten Klassifizierung durch die unabhängige Beschreibung einzelner Gesichtspunkte oder Facetten eines Dokuments (*Faceted Classification*) dagegen wird das Klassifizierungsschema in einem Bottom-Up-Ansatz durch die Analyse einer repräsentativen Auswahl der zu klassifizierenden Dokumente entwickelt. Die resultierenden *elementaren* Klassen können z. B. über hierarchische Beziehungen miteinander verbunden sein und werden auch als *Terme* bezeichnet. Zusammengehörige Gruppen solcher Klassen, die einen gemeinsamen übergeordneten Gesichtspunkt beschreiben, heißen *Facetten*. Da einzelne elementare Klassen für die Klassifizierung eines Dokuments im Allgemeinen nicht die erforderliche Präzision zulassen, wird ein Dokument bei der Faceted Classification in eine *zusammengesetzte* Klasse eingeordnet, die durch die Kombination elementarer Klassen aus verschiedenen Facetten bedarfsgerecht „synthetisiert“ wird.

Prieto-Diáz et al. stellen in [PDF87] ein Verfahren zur Klassifizierung von Komponenten vor, das auf dem Ansatz der Faceted Classification basiert und neben dem Auffinden von Komponenten auch die Bewertung des für ihren Einsatz erforderlichen Anpassungsaufwands unterstützt. Ein Klassifizierungsschema spannt bei diesem Verfahren einen multidimensionalen Suchraum auf, bei dem jede Dimension durch eine Facette und die ihr zugeordneten Terme definiert ist. Das betrachtete Klassifizierungsschema berücksichtigt dabei zum einen die Funktionalität einer Komponente durch Facetten zur Beschreibung der unterstützten Funktion, des manipulierten Objekts und des Mediums, auf dem die Funktion ausgeführt wird, sowie zum anderen die Einsatzumgebung einer Komponente durch Facetten zur Beschreibung des durch eine Komponente realisierten Systemtyps, ihres Funktionsbereichs und der Anwendungsdomäne. Eine Komponentenbeschreibung besteht aus einem Tupel von Termen, wobei jeder Term als Attributwert einer gewählten Facette (oder als Koordinate der durch sie repräsentierten Dimension) verstanden werden kann. Suchanfragen bestehen analog zu diesem Beschreibungsformat ebenfalls aus einem Tupel von Termen, die aus den einzelnen Facetten ausgewählt werden.

Da im Allgemeinen nicht von einer exakten Übereinstimmung einer Suchanfrage mit einer Komponentenbeschreibung ausgegangen werden kann, schlagen Prieto-Diáz et al. die Anordnung der Terme einer Facette in einem konzeptuellen Graphen vor, auf dem eine Metrik für den Abstand der Terme definiert ist. Auf der Grundlage dieser Metrik können Anfragen, die keine exakt passenden Suchergebnisse liefern, automatisch umformuliert werden, um auch Komponenten zu finden, die eng mit den ursprünglichen Anforderungen verwandt sind. Um mehrdeutige Beschreibungen zu vermeiden, setzen sie darüber hinaus ein kontrolliertes Vokabular von Termen ein, bei dem synonyme Terme unter einem einzelnen abstrakten Konzept gruppiert werden.

Der Ansatz der Faceted Classification fällt in die Kategorie der deskriptiven Suchverfahren. Als Vorteile der Faceted Classification gegenüber der weiter verbreiteten Enu-

merative Classification nennen Prieto-Díaz et al. die einfache Erweiterbarkeit des Klassifizierungsschemas um neue Facetten und Terme sowie die gesteigerte Präzision und Flexibilität bei der Klassifizierung.

4.3.2 LaSSIE – Large Software System Information Environment

Devanbu et al. [DBSB91] haben mit *LaSSIE (Large Software System Information Environment)* ein System zur Dokumentation von Architektur und Komponenten eines komplexen Softwaresystems aus dem Telekommunikationsbereich entwickelt. Als Komponenten werden dabei einzelne Operationen des Systems betrachtet. LaSSIE umfasst eine umfangreiche Wissensbasis über dieses Softwaresystem, ein semantisches Suchverfahren auf der Grundlage formaler, klassifizierender Schlussfolgerungen sowie eine Benutzungsschnittstelle mit einem graphischen Browser und einem Parser für natürlichsprachliche Anfragen.

Grundlage der Wissensrepräsentation in LaSSIE ist die Wissensrepräsentationssprache *KANDOR* [PS84], die Wissen in Form von *Frames* darstellt. Ein Frame stellt eine komplexe Beschreibung einer Klasse dar, indem er Bedingungen spezifiziert, die von allen Objekten der Klasse (den *Individuen* des Frames) erfüllt werden müssen. Diese Bedingungen werden in Form so genannter *Slots* formuliert, die Beziehungen eines Individuums des Frames zu anderen Individuen beschreiben. Frames sind in einer Generalisierungs-/Spezialisierungshierarchie angeordnet und definieren somit eine Taxonomie von Klassen, deren Wurzel die allgemeinste Klasse `THING` bildet. Die vollständige Beschreibung eines Individuums ergibt sich somit aus dem Frame, dem das Individuum angehört, sowie der Menge aller generelleren Frames und den Beziehungen zu anderen Individuen.

Die grundlegende Struktur der Wissensbasis von LaSSIE ist durch vier grundlegende Klassen `DOER`, `OBJECT`, `ACTION` und `STATE` gegeben, die unterhalb der allgemeinsten Klasse `THING` angesiedelt sind und durch weitere domänenspezifische Klassen spezialisiert werden. `DOERs` haben die Fähigkeit, Aktionen (`ACTIONs`) auszuführen, denen Objekte (`OBJECTs`) zugeordnet sein können. Ein `STATE` beschreibt den Zustand einer Klasse (z. B. eines `DOERs` oder `OBJECTs`) vor oder nach Ausführung einer solchen Aktion. Eine Komponentenbeschreibung entspricht in LaSSIE einem Individuum eines speziellsten `ACTION`-Frames der Wissensbasis und dessen spezifische Verknüpfung mit `DOERs`, `OBJECTs` und `STATEs`. Suchanfragen an LaSSIE werden analog dazu durch die Spezifikation eines `ACTION`-Frames formuliert. Die Suche nach geeigneten Komponenten erfolgt dann, indem die Anfrage durch den Klassifikationsmechanismus von KANDOR in die Taxonomie eingeordnet wird und alle Individuen der spezielleren Frames bestimmt werden.

LaSSIE bietet eine natürlichsprachliche Schnittstelle für die Formulierung von Suchanfragen. Grundlage dieser Schnittstelle sind ein Lexikon relevanter Begriffe des Anwendungsbereichs, eine Taxonomie dieser Begriffe sowie eine Menge von „Kompatibilitätstupeln“, die die Kombinationsmöglichkeiten dieser Begriffe zur Formulierung von Anfragen reglementieren. Die Begriffe der natürlichsprachlichen Schnittstelle sind den Frames der Wissensbasis zugeordnet. Die Struktur der Frames wird in natürlichsprachlichen Anfragen durch geeignete Präpositionen wiedergegeben.

LaSSIE kann aufgrund dieser natürlichsprachlichen Orientierung als Information-Retrieval-Verfahren betrachtet werden [MMM98]. Der Einsatz der Klassifizierung als grund-

legendes Suchprinzip legt allerdings auch eine Einordnung als deskriptives Verfahren nahe. Devanbu et al. nennen als Mängel von LaSSIE die fehlende Möglichkeit, Ganzes/Teile-Beziehungen zwischen Konzepten repräsentieren zu können, sowie Schwächen hinsichtlich der Präzision der Komponentenbeschreibungen. Als Vorteile erwähnen sie, dass LaSSIE als klassifizierendes Verfahren auch solche Komponenten findet, die der Anfrage nicht exakt entsprechen.

4.3.3 ROSA – Reuse Of Software Artifacts

Grundlage des von Girardi et al. im Rahmen des Projekts *ROSA (Reuse Of Software Artifacts)* [GI94a, GI94b] verfolgten Ansatzes zur Klassifikation und Suche von Komponenten ist die Verarbeitung natürlichsprachlich formulierter Komponentenbeschreibungen und Suchanfragen. ROSA umfasst ein Klassifikationssystem, mit dem Komponenten durch die automatische Extraktion lexikalischer, syntaktischer und semantischer Informationen aus natürlichsprachlichen Beschreibungen indexiert werden, sowie ein Suchverfahren, das auf der Feststellung semantischer Ähnlichkeiten zwischen einer Suchanfrage und einer Komponentenbeschreibung basiert.

Die Klassifizierung einer Komponente wird durch die lexikalische, syntaktische und semantische Analyse ihrer natürlichsprachlichen Beschreibung sowie die Überführung der Analyseergebnisse in eine interne frameähnliche Repräsentation erreicht. In der lexikalischen Analyse, für die das Lexikon *WordNet* [Mil95] eingesetzt wird, werden z. B. Wörter auf ihre Stammform reduziert und ihre grammatische Kategorie erkannt. Die syntaktische und semantische Analyse basieren auf einer Teilmenge der Grammatik für englische Aussagesätze. In der semantischen Analyse werden die semantischen Fälle bestimmt, über die die Substantive eines Aussagesatzes mit dessen Prädikat verbunden sind. Für die eigentliche Klassifizierung einer Komponente steht ein Klassifizierungsschema zur Verfügung, das auf der Grundlage der verschiedenen semantischen Fälle eine Hierarchie *generischer* Frames definiert. Generische Frames modellieren die semantischen Strukturen verschiedener Formen von Aussagesätzen sowie zusätzliche Informationen zu einer Komponente wie z. B. ihren Quellcode oder Attribute zur Unterstützung ihrer Wiederverwendung. Die Ergebnisse der Analyse einer Komponentenbeschreibung werden in Form von Frames repräsentiert, die als Spezialisierung generischer Frames in die Hierarchie eingeordnet werden. Dabei werden die semantischen Fälle eines Aussagesatzes als Slots eines (generischen) Frames repräsentiert.

Suchanfragen werden ebenso wie Komponentenbeschreibungen als Freitext formuliert und durch einen vergleichbaren Analysevorgang in eine frameähnliche Repräsentation überführt. Für die Suche nach geeigneten Komponenten bietet ROSA zwei Verfahren an, die auf der Bestimmung von Ähnlichkeiten zwischen den internen Repräsentationen einer Anfrage und einer Komponentenbeschreibung basieren. Das primär eingesetzte Verfahren bestimmt die Ähnlichkeit der zugrunde liegenden semantischen Strukturen in Abhängigkeit von der Anzahl einander entsprechender semantischer Fälle und dem lexikalischen Abstand der in diesen Fällen verwendeten Wörter. Der lexikalische Abstand zweier Wörter wird dabei durch die Betrachtung der Entfernung dieser Wörter im Lexikon bestimmt, wobei die lexikalischen Beziehungen Synonymie, Hyponymie (Spezialisierung) und Hyperonymie (Generalisierung) berücksichtigt werden. Die Anzahl der in einem Suchvorgang

gelieferten Komponenten und damit die Präzision des Suchverfahrens wird über einen Schwellwert kontrolliert. Ein zweites, ergänzendes Suchverfahren basiert auf der Bestimmung von Übereinstimmungen zwischen den in Anfrage und Komponentenbeschreibung eingesetzten Substantiven.

Wir ordnen den mit ROSA verfolgten Ansatz als Information-Retrieval-Verfahren ein. Da eine Komponente dann als geeignet bewertet wird, wenn ein Maß für ihre Ähnlichkeit mit der Anfrage einen vorgegebenen Schwellwert überschreitet, kann ROSA allerdings auch als topologisches Verfahren kategorisiert werden [MMM98]. Girardi et al. heben als Vorteile des mit ROSA verfolgten Ansatzes die Kostenreduktion bei der Indexerstellung in einem Komponenten-Repository sowie die hohen, bei Experimenten mit der Suche nach UNIX-Kommandos gemessenen Werte für Präzision und Trefferquote hervor. Zudem findet ROSA auch Komponenten, die der gestellten Anfrage nur ungefähr entsprechen. Als Nachteil erwähnen sie die Abhängigkeit des Suchverfahrens von präzisen und umfassenden Komponentenbeschreibungen sowie von qualitativ hochwertigen Lexika und Thesauri.

4.3.4 Signature and Specification Matching of Software Components

In ihren Arbeiten zur Suche nach wiederverwendbaren Funktionen und Modulen verfolgen Zaremski et al. in [ZW93] zunächst den Gedanken des *Signature Matching*, bei dem die Zuordnung einer Funktion zu einer Anfrage auf der Betrachtung von Signaturinformationen beruht. Die Signatur einer Funktion ist dabei durch die Liste der Typen ihrer Eingabe- und Ausgabeparameter sowie die von ihr erzeugten Ausnahmesituationen (Exceptions) gegeben. Beim Signature Matching von Funktionen wird allgemein festgestellt, ob ein so genanntes Match-Prädikat bzgl. einer Anfrage, die durch die Angabe der Signatur einer gesuchten Funktion spezifiziert wird, und der Signatur einer existierenden Funktion erfüllt ist. Durch den Austausch des Match-Prädikats sowie die Anwendung von Transformationsoperatoren auf die Anfrage bzw. die Signatur der Funktion definieren Zaremski et al. verschiedene Formen des Signature Matching. Neben der Suche nach exakten Übereinstimmungen zwischen einer Anfrage und einer Funktion lassen sich dadurch auch Funktionen finden, die ungefähr, d. h. nach Umsortierung ihrer Parameter oder Spezialisierung bzw. Generalisierung ihrer Parametertypen der Anfrage entsprechen. Für die Suche nach Funktionsmodulen existieren analoge Formen des Signature Matching, die auf der Übereinstimmung der in einer Anfrage spezifizierten Funktionen mit den im jeweilig betrachteten Modul enthaltenen Funktionen und benutzerdefinierten Typen basieren.

Ein Defizit des Signature Matching besteht darin, dass in Gestalt der Signatur lediglich strukturelle Informationen von Anfragen und Funktionen betrachtet und semantische Aspekte außer Acht gelassen werden. Aus diesem Grund ist es z. B. möglich, dass bei der Suche nach einer Funktion zur Addition zweier Integer-Werte durch die Spezifikation der Signatur $int \times int \rightarrow int$ eine Funktion zur Subtraktion zweier Integer-Werte gefunden wird. Zaremski et al. erweitern das Signature Matching daher beim *Specification Matching* [ZW97] um die Betrachtung der denotationellen Semantik. Die Beschreibung von Anfragen und Funktionen durch die Angabe von Signaturen wird dabei um die Spezifikation von Vor- und Nachbedingungen ergänzt. Ausgehend von der Äquivalenz der

Vor- und Nachbedingungen von Anfrage und Funktion definieren Zaremski et al. eine Hierarchie abgeschwächter Formen der semantischen Übereinstimmung. Diese Ausprägungen des Specification Matching gestatten u. a. die Verfeinerung einer Anfrage durch eine Funktion, indem im Sinne des Behavioural Subtyping durch Abschwächung der Vorbedingung und Verschärfung der Nachbedingung ein verhaltensmäßiger Subtyp bestimmt wird. Die Anforderungen an die semantische Übereinstimmung einer Funktion mit einer Anfrage werden beim Specification Matching jeweils in Gestalt von Formeln über den Vor- und Nachbedingungen spezifiziert, deren Beweisbarkeit Voraussetzung für die Zuordnung einer Funktion zu einer Anfrage ist. Analog zum Signature Matching unterstützt auch das Specification Matching die Suche nach Funktionsmodulen auf der Grundlage der Übereinstimmung einer Anfrage mit den enthaltenen Funktionen und Typen.

Das Specification Matching stellt ein denotationelles Suchverfahren dar. Da das Signature Matching als spezielle Form des Specification Matching betrachtet werden kann, bei dem die Vor- und Nachbedingungen durch das Prädikat *true* definiert sind, lässt es sich ebenfalls als denotationelles Verfahren auffassen [MMM98]. Zaremski et al. stellen in [ZW97] lediglich die Anwendung des Specification Matching auf relativ einfache Funktionen zur Manipulation von Datenstrukturen wie Listen oder Stacks vor. Die Anwendbarkeit dieses Ansatzes auf Komponenten mit komplexer fachlicher Semantik erscheint fragwürdig, da sowohl die Formulierung von Anfragen und Komponentenbeschreibungen als auch die Überprüfung ihrer Übereinstimmung einen hohen Aufwand verursachen.

Suchtechniken auf der Grundlage formaler Spezifikationen, wie sie Zaremski et al. mit dem Specification Matching vorgeschlagen haben, erfordern das (automatische) Beweisen einer Vielzahl komplexer logischer Formeln. Um den praktischen Einsatz solcher formalen Suchtechniken trotz der hohen Berechnungskomplexität zu ermöglichen, schlagen Fischer et al. [FSS98, Fis01] mit ihrem Werkzeug *NORA/HAMMR (Highly-Adaptive Multi-Method Retrieval Tool)* eine Pipeline-Architektur vor, bei der die zu analysierenden Komponenten eine Reihe von Filtern mit zunehmender deduktiver Mächtigkeit durchlaufen müssen. Mit diesen Filtern wird versucht, ungeeignete Komponenten möglichst frühzeitig im Suchprozess mit Hilfe einfacherer formaler Techniken zu eliminieren und komplexe formale Verfahren zum Beweisen von Formeln (*Theorem Proving*) nur auf sorgfältig gefilterte Komponenten anzuwenden.

4.3.5 Behaviour Sampling

Der Ansatz des *Behaviour Sampling*, den Podgurski et al. in [PP93] zur Suche nach einzelnen Operationen verfolgen, basiert direkt auf der durch die Ausführung einer Operation erzielten Transformation von Eingabedaten in Ausgabedaten und ist daher als Suchverfahren auf Grundlage der operationellen Semantik einzuordnen. Suchanfragen werden dabei in Form einer Schnittstellenspezifikation, eines Wertebereichs für die Eingabedaten, einer operationellen Eingabedatenverteilung sowie der jeweils erwarteten Ausgabedaten spezifiziert. Eine operationelle Eingabedatenverteilung (*operational input distribution*) definiert dabei die Wahrscheinlichkeit, mit der die gesuchte Operation in der Zielumgebung mit spezifischen Eingabedaten genutzt werden wird. Im Suchprozess wird für jede Operation, deren Schnittstelle die Schnittstellenspezifikation erfüllt, unter Berücksichtigung der Eingabedatenverteilung eine Anzahl von Eingabedatensätzen ausgewählt, mit

denen die Operation ausgeführt wird. Diejenigen Operationen, die die erwarteten Ausgabedaten liefern, werden in das Suchergebnis des Verfahrens aufgenommen.

Die zentrale Annahme des Behaviour Sampling besteht darin, dass eine Operation, die die Anforderungen eines Verwenders bzgl. zufällig gewählter Eingabedaten erfüllt, wahrscheinlich auch bei Betrachtung aller Anforderungen relevant ist. Die Präzision des Verfahrens kann natürlich nicht garantiert werden, da eine Operation, die sich auf den gewählten Eingabedaten wie erwartet verhält, auf anderen Daten ein erheblich von den Erwartungen abweichendes Verhalten zeigen kann. Da allerdings die Wahrscheinlichkeit, dass sich eine nicht relevante Operation bzgl. n Eingabedatensätzen erwartungsgemäß verhält, exponentiell mit n abnimmt, kann in der Praxis bereits mit kleinen n eine hohe Präzision erzielt werden. Die Trefferquote des Behaviour Sampling kann kontrolliert werden, indem die Anforderung an die Übereinstimmung von spezifizierten und erzielten Ausgabedaten (z. B. mittels eines „Orakels“) abgeschwächt wird [MMM98]. Neben der Beschränkung auf ausführbare Komponenten muss als Schwachpunkt des Verfahrens die Ausrichtung auf solche Operationen gewertet werden, die eine Funktion mit einfachen Eingabe- und Ausgabedatentypen berechnen.

Aufbauend auf dem Ansatz des Behaviour Sampling schlägt Pozewaunig [Poz01] zwei Verfahren zur Unterstützung der Komponentensuche vor, die auf der automatischen Generierung von Komponentenbeschreibungen durch die Analyse von Testdaten basieren. Zustandslose Komponenten, unter denen Pozewaunig einzelne Operationen versteht, werden in dem ersten Verfahren zunächst durch eine Technik namens *Generalized Signature Matching* auf Partitionen eines Repositories verteilt, wobei eine Partition alle Operationen mit der gleichen bzw. einer (z. B. aufgrund von Subtypeigenschaften) verträglichen Signatur enthält. Für jede dieser Partitionen wird dann durch das so genannte *Static Behaviour Sampling* ein Entscheidungsbaum erzeugt, der eine Navigationsstruktur auf dem Repository definiert. Grundlage der Induktion eines solchen Entscheidungsbaums sind charakteristische oder beispielgebende Testdaten (*Data Points*), anhand derer zwischen den zu beschreibenden Komponenten differenziert werden kann. Ein Verwender sucht nach einer zustandslosen Komponente, indem der die gewünschte Signatur spezifiziert und anschließend in einem Frage-Antwort-Zyklus durch schrittweise Angabe des gewünschten Verhaltens den Entscheidungsbaum herabsteigt, bis er an dessen Blättern eine (oder mehrere) Komponenten mit dem geforderten Verhalten findet.

In dem zweiten Suchverfahren betrachtet Pozewaunig zustandsbehaftete Komponenten, die im Sinne eines Moduls mehrere Operationen umfassen. Das Verhalten zustandsbehafteter Komponenten hängt im Gegensatz zu dem zustandsloser Komponenten nicht allein von den Eingabedaten, sondern auch vom internen Zustand der Komponente ab. Testdaten bestehen folglich nicht mehr aus einfachen Paaren von Eingabe- und Ausgabedaten, sondern vielmehr aus zulässigen Sequenzen von Operationsaufrufen. Das *Abstracted Behaviour Sampling* abstrahiert von den konkreten Werten, mit denen einzelne Operationen in diesen Testfällen aufgerufen werden, und erzeugt eine operationelle Repräsentation des Verhaltens einer Komponente. Auf der Grundlage der Algorithmen *kTail* bzw. *SEQUITUR* werden dabei aus den in den Testdaten einer Komponente spezifizierten Operationssequenzen endliche Automaten bzw. kontextfreie Grammatiken generiert, die das Verhalten der Komponente beschreiben. Im Gegensatz zum Static Behaviour Sampling, bei dem das Verhalten einer Komponente die Grundlage für die Navigation durch

das Repository bildet, unterstützt das Abstracted Behaviour Sampling die Komponentensuche lediglich durch die Erzeugung einer operationellen Beschreibung des Verhaltens einer Komponente, die den Verwender beim „manuellen“ Vergleich seiner Anforderungen mit der Funktionalität einer Komponente unterstützt.

4.3.6 Refinement-Based Component Storage and Retrieval

Mili et al. verfolgen in ihrer Arbeit zur Speicherung und Suche von Komponenten [MMM94] einen dem Behaviour Sampling ähnlichen Ansatz zur Spezifikation des funktionalen Verhaltens, wobei als Komponenten wiederum einzelne Operationen betrachtet werden. Beim *Refinement-Based Retrieval* besteht eine funktionale Spezifikation einer Komponente aus einer binären Relation „korrekter“ Paare von Eingabe- und Ausgabedaten, die mittels logischer Prädikate beschrieben ist. Das zentrale Prinzip zur Organisation von Komponenten in einem Repository und zu deren effizienter Suche besteht in der Verwaltung einer *Verfeinerungsrelation* zwischen diesen Spezifikationen sowie einer *Korrektheitsrelation* zwischen den Spezifikationen und den verwalteten Komponenten. Die Verfeinerung einer Spezifikation s durch eine Spezifikation s' drückt dabei aus, dass jede Komponente, die die Spezifikation s' erfüllt, auch eine Lösung von s darstellt. Verfeinerungsbeziehungen werden dabei mittels eines Werkzeugs zum Theorem Proving bestimmt. Über die Korrektheitsrelation sind einer Spezifikation alle Komponenten zugeordnet, die bzgl. der Spezifikation korrekt sind, jedoch keine ihrer Verfeinerungen erfüllen.

Die Suche nach einer Komponente wird durch die Formulierung einer Anfrage in Form einer funktionalen Spezifikation eingeleitet, die in das Verfeinerungsgitter der im Repository verwalteten Spezifikationen verfügbarer Komponenten eingeordnet wird. Über die Bestimmung aller Spezifikationen, die die Anfrage verfeinern, lassen sich dann mit Hilfe der Korrektheitsrelation die bzgl. der Anfrage korrekten Komponenten ermitteln. Neben diesem exakten Suchverfahren definieren Mili et al. auch ein Verfahren zur Suche nach annähernd passenden Komponenten, das auf der Minimierung eines Maßes für den Abstand zwischen den Spezifikationen einer Anfrage und einer Komponente basiert. Der Abstand einer Anfrage von einer Komponente wird dabei durch die Ermittlung der Relation gemessen, die genau die Informationen enthält, in denen Anfrage und Komponente übereinstimmen. Da eine Verfeinerungsbeziehung zwischen zwei Relationen impliziert, dass die verfeinernde Relation mehr (übereinstimmende) Informationen enthält als die verfeinerte Relation, ist eine Ordnung auf diesen „Übereinstimmungsrelationen“ und damit auf dem Abstand der Anfrage von den betrachteten Komponenten wiederum durch die Verfeinerungsrelation definiert.

Der Ansatz von Mili et al., der anders als das Behaviour Sampling durch die Verwendung von formalen Spezifikationen des Verhaltens auch auf nicht ausführbare Artefakte wie z. B. Entwürfe anwendbar ist, stellt ein Suchverfahren auf der Grundlage der denotationellen Semantik dar. Da allerdings auch die Suche nach annähernd passenden Komponenten unterstützt wird, ist darüber hinaus eine Einordnung als topologisches Verfahren sinnvoll. Positiv hervorzuheben ist bei diesem Ansatz, dass eine Anfrage lediglich einmal klassifiziert werden muss und somit individuelle Vergleiche mit den verfügbaren Komponenten entfallen. Als Schwachpunkte des Verfahrens sind allerdings die hohe

Komplexität des Theorem Proving sowie der formale Charakter von Komponentenbeschreibungen und Suchanfragen zu nennen.

4.3.7 OntoSeek – Ontology-Based Information Retrieval

Guarino et al. haben mit *OntoSeek* [GMV99] ein Werkzeug zum ontologiegestützten Information Retrieval im Kontext spezialisierter Repositories wie z. B. Produktkatalogen entwickelt. OntoSeek kombiniert die Nutzung einer linguistischen Ontologie, die die Entkopplung des Vokabulars eines Benutzers von dem des Repositories gestattet, mit einem strukturierten Wissensrepräsentationsformalismus eingeschränkter Ausdrucksmächtigkeit, der zur Erhöhung von Präzision und Trefferquote gegenüber einfachen Suchverfahren auf der Grundlage von Schlüsselwörtern und Attribut/Wert-Paaren beiträgt. Die Komponentensuche mit OntoSeek stellt einen Spezialfall des ontologiegestützten Information Retrieval dar [BGMV97].

Das grundlegende Prinzip der Komponentensuche mit OntoSeek besteht in der Bestimmung von Subsumtionsbeziehungen zwischen Anfragen und Komponentenbeschreibungen, die jeweils in Gestalt konzeptueller Graphen mit beschrifteten Knoten und Kanten formuliert werden. Die durch eine Komponente angebotene Funktionalität wird dabei durch einen so genannten *Lexical Conceptual Graph*[1] spezifiziert, dessen Knoten und Kanten mit englischen Wörtern beschriftet sind. Dabei drückt eine Kante zwischen zwei Knoten eine durch ihre Beschriftung näher bestimmte Beziehung zwischen den durch die Knoten repräsentierten Konzepten bzw. Individuen aus. Um evtl. vorhandene Mehrdeutigkeiten hinsichtlich der Wortwahl aufzulösen, wird dieser „Graph von Wörtern“ mittels *Sensus* [KL94] interaktiv in einen „Graph von Bedeutungen“ (*Semantic Graph*) übersetzt. Sensus ist eine linguistische Ontologie, die eine einfache taxonomische Struktur über einem englischen Wortschatz definiert und u. a. aus *WordNet* hervorgegangen ist. Durch Zugriff auf diese taxonomischen Beziehungen besteht neben der Auswahl von Bedeutungen die Möglichkeit, gewählte Bedeutungen weiter zu verfeinern. Analog zur Erstellung einer Komponentenbeschreibung ist auch für die Formulierung einer Anfrage ein solcher semantischer Graph zu spezifizieren. Eine Komponente erfüllt eine Anfrage, wenn eine Subsumtionsbeziehung zwischen den entsprechenden semantischen Graphen hinsichtlich der Struktur und der Beschriftung der Knoten und Kanten festgestellt werden kann. Etwas präziser formuliert subsumiert ein Graph Q zur Spezifikation einer Anfrage einen Graphen C zur Beschreibung einer Komponente, wenn Q isomorph zu C ist, die Beschriftungen der Knoten und Kanten von Q durch entsprechende Beschriftungen von C spezialisiert werden und der Kopf von Q dem Kopf von C entspricht.

Das in OntoSeek eingesetzte Suchverfahren stellt eine erweiterte Form deskriptiver Suche dar. Guarino et al. heben als Vorzüge von OntoSeek hervor, dass sie eine Wissensrepräsentationssprache beschränkter Ausdrucksstärke nutzen, die insbesondere die Anfrageformulierung vereinfacht, und den umfangreichen Wortschatz der natürlichen (englischen) Sprache für die Spezifikation von Komponenten und Anfragen zulassen, anstatt sich auf ein kontrolliertes Vokabular zu beschränken.

[1] In [BGMV97] wird stattdessen noch von *Lexical Semantic Graphs* gesprochen.

4.3.8 BALES – Binding Business Applications to LEgacy Systems

Im Gegensatz zu den bislang vorgestellten Arbeiten zur Komponentensuche schlägt VAN DEN HEUVEL mit *BALES (Binding Business Applications to LEgacy Systems)* [Heu02] eine Methodik zur Suche nach wiederverwendbaren Komponenten in Legacy-Systemen vor. Legacy-Systeme implementieren vielfach zentrale betriebliche Funktionen zur Unterstützung der Kern-Geschäftsprozesse eines Unternehmens, erfüllen allerdings oft nicht mehr die Anforderungen, die an moderne Informationssystemarchitekturen z. B. hinsichtlich Flexibilität, Schnittstellen oder Verteilung gestellt werden. Das Ziel von BALES besteht daher darin, wiederverwendbare Legacy-Komponenten aufzuspüren und in moderne betriebliche Informationssysteme zu integrieren.

Die *BALES Roadmap* sieht als Vorgehensmodell drei Phasen vor, an deren Ende die Parametrisierung eines Informationssystems mit geeigneten Legacy-Komponenten steht. In der *Forward Engineering Phase* wird dabei zunächst ein integriertes Unternehmensmodell in Form einer Sammlung kollaborierender Geschäftsobjekte entwickelt, das als Anforderungsdefinition dient. Grundlage der Spezifikation dieses Unternehmensmodells ist mit der *Component Definition Language (CDL)* eine Komponentenbeschreibungssprache, die Bestandteil der inzwischen nicht mehr weiter entwickelten *Business Object Component Architecture (BOCA)* [OMG98] ist. In der anschließenden *Reverse Engineering Phase* werden Wrapper-Spezifikationen der verfügbaren Legacy-Komponenten auf analoge Weise mittels CDL beschrieben. Ziel der *Linking Phase* schließlich ist die Parametrisierung von Geschäftsobjekten mit solchen Legacy-Komponenten, die die Aufgaben des Geschäftsobjekts durch ihre Funktionalität unterstützen helfen. Dazu wird im Rahmen des *Semantic Content Matching* zunächst die semantische Ähnlichkeit der Spezifikation eines Geschäftsobjekts mit Wrapper-Spezifikationen von Legacy-Komponenten ermittelt. Die inhaltliche Semantik einer CDL-Spezifikation wird dabei definiert, indem die z. B. zur Benennung von Operationen und Attributen verwendeten Bezeichner auf ein gemeinsames Domänenmodell (eine Terminologie) abgebildet werden. In BALES kommt dabei *WordNet* zum Einsatz, obwohl grundsätzlich auch die Verwendung anderer Ontologien mit entsprechenden Eigenschaften denkbar ist. Die inhaltliche Nähe zweier CDL-Spezifikationen lässt sich dann in Form des Winkels zwischen zwei Vektoren bestimmen, die die zu vergleichenden CDL-Spezifikationen in einem durch die verwendete Terminologie aufgespannten multidimensionalen Vektorraum repräsentieren. Beim Semantic Content Matching werden alle Legacy-Komponenten als relevant angesehen, deren semantische Ähnlichkeit mit dem Geschäftsobjekt einen vorgegebenen Schwellwert übersteigt. Mit dem anschließenden *Object Model Matching* wird die strukturelle Ähnlichkeit zwischen der Spezifikation eines Geschäftsobjekts mit Wrapper-Spezifikationen von Legacy-Komponenten bewertet, indem aus den jeweiligen CDL-Spezifikationen Typ-Bäume abgeleitet werden, die unter Berücksichtigung verschiedener Anforderungen an die Kompatibilität von Typen miteinander verglichen werden. Typ-Bäume beschreiben dabei die Datentypen der strukturellen Eigenschaften von Geschäftsobjekten und Legacy-Komponenten wie z. B. Attribute und Assoziationen sowie die Signaturen ihrer Operationen. Das tatsächliche, z. B. mittels Zustandsmaschinen festgehaltene Verhalten wird beim Object Model Matching nicht berücksichtigt. Schließlich werden die beim Semantic Content und Object Model Matching ermittelten Ähnlichkeitsmaße durch einen Entwickler bewertet und

– sofern eine geeignete Legacy-Komponente gefunden wurde – evtl. vorhandene Typ-Konflikte aufgelöst, bevor die Legacy-Komponente in das Geschäftsobjekt eingebunden wird.

Aufgrund der Bestimmung eines (Mindest-)Maßes für die Ähnlichkeit von Geschäftsobjekt und Legacy-Komponenten ist der in BALES verfolgte Ansatz zur Komponentensuche als topologisches Verfahren einzuordnen. Die Tatsache, dass Komponenten in BALES allgemein als konzeptuelle Graphen spezifiziert werden, sowie die Benutzung von Algorithmen des klassischen Information Retrieval legen darüber hinaus auch eine Einordnung als (erweitertes) deskriptives Verfahren bzw. Verfahren des Information Retrieval nahe.

4.4 Bewertung

Eine kritische Bewertung von Ansätzen zur Komponentensuche ist im Allgemeinen durch die Erfahrungen und die Erwartungshaltung des Kritikers gefärbt. Da wir uns im Rahmen unserer Arbeit auf die geschäftsprozessorientierte Suche nach Fachkomponenten auf Komponentenmärkten konzentrieren, ist unsere Perspektive auf die in den Abschnitten 4.2 und 4.3 vorgestellten Ansätze zur Komponentensuche durch die Forderung nach angemessener Berücksichtigung fachlicher Aspekte geprägt.

Wie in Abschnitt 4.2 dargelegt wurde, ist die Praxis der Komponentensuche auf aktuellen Komponentenmärkten zumeist durch simple Suchverfahren wie Stichwortsuche in Freitextbeschreibungen, einfache deskriptive Verfahren oder die Navigation durch Kategorien gekennzeichnet. Als Kritik an diesen Verfahren ist zunächst festzuhalten, dass die mit ihnen verbundenen Anfragesprachen eine angemessene Formulierung des Situationsmodells eines Verwenders in einer Suchanfrage grundsätzlich ausschließen. Lediglich die vereinzelt angebotene Möglichkeit zur Bewertung spezifischer Anforderungen in umfangreichen, fachlich ausgerichteten Kriterienkatalogen unterstützt die Spezifikation des Situationsmodells. Allerdings lassen auch diese Suchverfahren die Betrachtung von Prozesswissen, das in Form von Geschäftsprozessmodellen festgehalten wurde, vermissen. Einhergehend mit den mangelhaften Möglichkeiten zur Spezifikation des Situationsmodells ist als zweiter Kritikpunkt die niedrige Präzision der eingesetzten Suchverfahren festzuhalten. Der Einsatz „intelligenterer“ Suchverfahren, die zwecks Steigerung der Präzision auch komplexere semantische Zusammenhänge berücksichtigen, würde allerdings die Verwendung präziserer Komponentenbeschreibungen voraussetzen. In diesem Zusammenhang ist schließlich zu kritisieren, dass die derzeit auf Komponentenmärkten eingesetzten Sprachen zur Beschreibung von Komponenten weder die angemessene Abbildung des Systemmodells eines Herstellers gestatten noch die präzise, an fachlichen Merkmalen ausgerichtete Komponentensuche durch Umfang, Detaillierungsgrad und Einheitlichkeit unterstützen.

In der Forschung auf dem Gebiet der Komponentensuche sind eine Vielzahl von Ansätzen untersucht worden, mit denen die vielfältigen Anforderungen der Komponentensuche an die eingesetzten Suchverfahren, aber auch an die Formulierung des Situations- und Systemmodells in unterschiedlichsten Kontexten erfüllt werden sollen. In Abschnitt 4.3 haben wir exemplarisch einige Vertreter dieser Ansätze vorgestellt, die in Tabelle 4.1 zusammengefasst werden. Hinsichtlich einer Bewertung von Präzision und Trefferquote kommen Mili et al. in ihrer Untersuchung alternativer Suchverfahren [MMM98] zu dem

Kriterium / Ansatz	Kategorie	Beschreibungsgegenstand	Beschreibungs- bzw. Anfrageform	Suchprinzip	Kontroll. Vokabular
Faceted Classification	• deskriptiv	Funktionalität/ Einsatzumgebung	Schlüsselwörter/Facetten	konzeptueller Abstand von Begriffen	√
LaSSIE	• deskriptiv • Inf. Retrieval	Funktionalität	konzeptueller Graph/ linguistische Strukturen	Subsumtionsrelation auf Graphen/Wörtern	√
ROSA	• Inf. Retrieval • topologisch	unspezifisch	Freitext	Ähnlichkeit semantischer Strukturen mit lexikalisch verwandten Wörtern	√ (WordNet)
Signature Matching	• denotationelle Semantik	Signaturen	Typspezifikation	strukturelle Ähnlichkeit (Typtransformation)	—
Specification Matching	• denotationelle Semantik	Verhalten	Vor- und Nachbedingungen	Theorem Proving	—
(Static) Behaviour Sampling	• operationelle Semantik	Verhalten	Ein-/Ausgabedaten (extensional)	Ausführung und Vergleich der Ausgabedaten	—
Abstr. Behaviour Sampling	• operationelle Semantik	Verhalten	Protokoll	— (Browsing/Review)	—
Refinement-Based Retrieval	• denotationelle Semantik • topologisch	Verhalten	Ein-/Ausgabedaten (intensional)	Theorem Proving (Verfeinerungs- und Korrektheitsrelation)	—
OntoSeek	• deskriptiv • Inf. Retrieval	Funktionalität	konzeptueller Graph	Subsumtionsrelation auf Graphen/Wörtern	√ (Sensus)
BALES	• topologisch • deskriptiv • Inf. Retrieval	Objektmodell	konzeptueller Graph	Ähnlichkeit von Begriffen/ Typkompatibilität	√ (WordNet)

Tabelle 4.1: Gegenüberstellung verschiedener Suchverfahren

Schluss, dass Suchverfahren auf der Grundlage der operationellen oder denotationellen Semantik die höchste Präzision erzielen. Eine ebenfalls hohe Präzision weisen daneben (erweiterte) deskriptive Verfahren auf. Für alle diese Verfahren gilt darüber hinaus, dass sie eine hohe Trefferquote erreichen. Neben Präzision und Trefferquote sind aus Sicht der geschäftsprozessorientierten Komponentensuche jedoch noch weitere Kriterien zu betrachten, die wir bei der folgenden kurzen Bewertung der in Abschnitt 4.3 vorgestellten Ansätze berücksichtigen.

Die Faceted Classification stellt ein einfaches Suchverfahren dar, das jedoch durch die Begrenzung der Facetten und Terme im Klassifizierungsschema einen engen Rahmen für die Beschreibung von Komponenten und die Formulierung von Anfragen steckt. Der in LaSSIE verfolgte Ansatz, das Wissen über Komponenten in Form natürlichsprachlicher Aussagen zu repräsentieren, kommt dem menschlichen Benutzer eines Komponenten-Repositories entgegen, indem es die Formulierung von Anfragen erleichtert und das Verständnis von Komponentenbeschreibungen unterstützt. Interessant erscheint außerdem der Gedanke, relevante Komponenten aufgrund von Spezialisierungsbeziehungen zwischen einer (generelleren) Anfrage und einer (spezielleren) Komponentenbeschreibung bzgl. dieser natürlichsprachlichen Aussagen zu bestimmen. ROSA gestattet durch die Verwendung der natürlichen (englischen) Sprache die größte Flexibilität bei der Beschreibung von Komponenten und der Formulierung von Anfragen. Allerdings wird diese Flexibilität damit erkauft, dass ROSA als Repräsentant der Information-Retrieval-Verfahren keinerlei Spezifika von (Komponenten-)Software berücksichtigt. Es erscheint daher fragwürdig, ob sich die hohen Präzisionswerte, die bei Experimenten mit der Suche nach UNIX-Kommandos ermittelt wurden, auch im Kontext der Suche nach Fachkomponenten mit komplexer Geschäftslogik wiederholen lassen.

Der Relevanzbegriff des Signature Matching stützt sich lediglich auf den Signaturen von Operationen ab. Da dieser Ansatz die Berücksichtigung jeglicher inhaltlicher Aspekte vermissen lässt, ist er für die Suche nach Fachkomponenten, die sich gerade durch ihren fachlichen Bezug auszeichnen, nicht geeignet. Für das Specification Matching gilt grundsätzlich, dass sich eine beliebig hohe Präzision hinsichtlich der syntaktischen und semantischen Übereinstimmung von Anfrage und Komponente erzielen lässt. Die Formulierung von Anfragen zur Repräsentation des Situationsmodells in Form logischer Prädikate erscheint allerdings für die Spezifikation fachlicher Anforderungen unangemessen. Zudem erreichen Suchanfragen und Komponentenbeschreibungen schnell eine logische Komplexität, die zum einen den menschlichen Benutzer überfordert, zum anderen aber auch den für das Theorem Proving anzusetzenden Aufwand in die Höhe treibt. Dieser Aufwand lässt sich lediglich im Kontext der Entwicklung sicherheitskritischer Systeme (vgl. z. B. [JM97]) rechtfertigen.

Als Vorteil des Behaviour Sampling ist die Möglichkeit zu bewerten, die Präzision der Suchergebnisse prinzipiell beliebig steigern zu können. Einschränkend muss jedoch angemerkt werden, dass die Spezifikation von Komponenten und Anfragen durch Tupel von Eingabe- und Ausgabedaten einen stark mathematisch geprägten Charakter aufweist. Während derartige Spezifikationen für die Suche nach Operationen in mathematischen Funktionsbibliotheken durchaus angemessen sein mögen, erscheint eine Spezifikation von fachlichen Anforderungen und Komponenten unmöglich. Das Static Behaviour Sampling definiert zwar eine Navigationsstruktur, die z. B. für die Suche in mathematischen

Funktionsbibliotheken geeignet erscheint, kann jedoch diese Einschränkung nicht aufheben. Das Abstracted Behaviour Sampling stellt kein Suchverfahren im eigentlichen Sinne dar, da es lediglich die „manuelle“ Bewertung einer gefundenen Komponente durch eine Komponentenbeschreibung unterstützt, die das Verhalten einer Komponente in Form eines Protokolls zulässiger Operationssequenzen spezifiziert. Beim Refinement-Based Retrieval wird der bereits bei LaSSIE verfolgte Ansatz aufgegriffen, relevante Komponenten aufgrund von Verfeinerungsbeziehungen zwischen Komponentenbeschreibungen und Suchanfragen zu identifizieren. Aus Sicht der Suche nach Fachkomponenten ist allerdings wie bereits beim Behaviour Sampling der mathematische Charakter der Komponentenbeschreibungen und Anfragespezifikationen zu bewerten, der die Formulierung fachlicher Anforderungen nicht praktikabel erscheinen lässt.

Bessere Unterstützung für die Beschreibung des Situationsmodells bietet OntoSeek durch den Ansatz, Anforderungen in Form konzeptueller Graphen zu spezifizieren, deren Knoten und Kanten mit einzelnen Wörtern bzw. deren Bedeutungen annotiert sind. Auch bei OntoSeek findet sich der bereits positiv bewertete Verfeinerungsgedanke wieder. An BALES schließlich ist der Gedanke, das Situationsmodell in Form eines mit Begriffen annotierten Objektmodells zu spezifizieren, positiv hervorzuheben. Allerdings richtet sich dieses Beschreibungsformat weniger an Fachexperten, sondern vielmehr an Technologieexperten wie z. B. Entwickler. Darüber hinaus erscheint der Vergleich von Typ-Bäumen im Sinne des Signature Matching eine für die Komponentensuche auf Komponentenmärkten zu starke Anforderung darzustellen.

Allgemein ist festzustellen, dass keines der vorgestellten Suchverfahren das Verhalten von Komponenten berücksichtigt, das sich im Sinne eines Protokolls aus der zustandsabhängigen Ausführbarkeit von Operationen ergibt. Zu begründen ist dieser Mangel damit, dass in der Mehrheit der vorgestellten Arbeiten einzelne Operationen als wiederverwendbare Komponenten betrachtet werden. Ein Komponentenbegriff, wie er in aktuellen Komponententechnologien anzufinden ist, wird lediglich von den aktuelleren Arbeiten zum Abstracted Behaviour Sampling sowie zu OntoSeek und BALES berücksichtigt. Das Abstracted Behaviour Sampling berücksichtigt den Protokollgedanken allerdings lediglich bei der Generierung von Komponentenbeschreibungen. OntoSeek verzichtet gänzlich auf die Berücksichtigung von Protokollinformationen bei der Komponentensuche, da ein zentrales Ziel des Ansatzes in der Verwendung einer Komponentenbeschreibungs- und Anfragesprache mittlerer Ausdrucksmächtigkeit besteht. Van den Heuvel schließlich betrachtet in BALES zwar ebenfalls keine Protokollinformationen, erkennt aber die Sinnhaftigkeit einer entsprechenden Erweiterung seines Ansatzes in seiner Dissertation [Heu02] an. Unterstützung erfährt der Gedanke, Protokollinformationen bei der Komponentensuche zu berücksichtigen, durch erste Ergebnisse einer empirischen Untersuchung, in der Kratz et al. die Ähnlichkeit von Signaturen und Protokollen funktional gleichwertiger Komponenten verglichen haben [KRH03].

4.5 Zusammenfassung

In diesem Kapitel haben wir den State of the Art auf dem Gebiet der Komponentensuche überblicksartig zusammengefasst. Dabei haben wir im Anschluss an eine grundlegende Einführung in das Problem der Komponentensuche und die Vorstellung verschiedener

Kategorien von Suchverfahren zunächst den praktischen Einsatz von Verfahren zur Komponentensuche auf aktuellen Komponentenmärkten skizziert. Ein Schwerpunkt dieses Kapitels lag auf der Darstellung des Spektrums der Forschungsarbeiten auf dem Gebiet der Komponentensuche, wobei wir mit Ausnahme der strukturellen Verfahren für jede der zuvor eingeführten Kategorien zumindest einen repräsentativen Ansatz näher vorgestellt haben. In der abschließenden Bewertung haben wir die Defizite der Praxis aktueller Komponentenmärkte herausgestellt sowie Vor- und Nachteile der einzelnen Forschungsansätze diskutiert. Als zentralen Mangel haben wir dabei die Vernachlässigung des Verhaltens von Komponenten identifiziert, das in Gestalt von Protokollen durch die zustandsabhängige Bereitstellung von Operationen definiert ist. Positiv festzuhalten ist neben dem Einsatz von Beschreibungsformen mit natürlichsprachlichem Charakter der Gedanke, Komponenten auf der Grundlage von Verfeinerungsbeziehungen zwischen einer (generelleren) Anfrage und einer (spezielleren) Komponentenbeschreibung zu suchen.

Teil II

Geschäftsprozessorientierte Komponentensuche: Anforderungen und Konzepte

Kapitel 5

Geschäftsprozessorientierte Komponentensuche im Überblick

In diesem Kapitel geben wir einen einleitenden Überblick über den Ansatz der geschäftsprozessorientierten Komponentensuche, den wir in den nachfolgenden Kapiteln des zweiten Teils dieser Arbeit konkretisieren. Wir umreißen dazu in Abschnitt 5.1 zunächst Ziele und Gegenstand der geschäftsprozessorientierten Komponentensuche, bevor wir in Abschnitt 5.2 dieses Kapitels einen einfachen, abstrakten Makroprozess vorstellen, der den grundlegenden Ablauf der geschäftsprozessorientierten Komponentensuche auf Komponentenmärkten skizziert. Im anschließenden Abschnitt 5.3 motivieren wir den Einsatz des Behavioural Subtyping als Kernkonzept geschäftsprozessorientierter Komponentensuche. Auf dieser Grundlage konkretisieren wir in Abschnitt 5.4 den Makroprozess der geschäftsprozessorientierten Komponentensuche und geben eine Übersicht über die folgenden Kapitel von Teil II. Wir beschließen das Kapitel in Abschnitt 5.5 mit einer kurzen Zusammenfassung.

5.1 Ziele und Gegenstand geschäftsprozessorientierter Komponentensuche

Die Ziele der geschäftsprozessorientierten Komponentensuche ergeben sich unmittelbar aus den in Kapitel 4 identifizierten Defiziten der in Praxis und Forschung eingesetzten Suchverfahren und Beschreibungstechniken. Ein erstes Ziel besteht dementsprechend in der stärkeren Berücksichtigung der Rolle von Fachexperten bei der Komponentensuche durch die Bereitstellung einer Anfragesprache, mit der insbesondere die Formulierung fachlicher Anforderungen, die sich vielfach an Geschäftsprozessen orientieren, unterstützt wird. Ein zweites Ziel der geschäftsprozessorientierten Komponentensuche ist durch die Forderung nach einer Komponentenbeschreibungssprache gegeben, die die möglichst umfassende Spezifikation einer Komponente gestattet und aufgrund der Berücksichtigung technischer und fachlicher Aspekte gleichermaßen für Technologie- und Fachexperten verständlich ist. Schließlich besteht ein drittes Ziel in der Konzeption eines Suchverfahrens, das das in Suchanfragen festgehaltene Prozesswissen bei der Komponentensuche berücksichtigt. Die für die Bewertung von Präzision und Trefferquote eines Suchverfahrens wich-

tige Relevanz einer Komponente ergibt sich dabei aus der Fähigkeit einer Komponente, die Ausführung der einzelnen Aktivitäten eines Geschäftsprozesses durch die bedarfsgerechte Bereitstellung von Diensten mit geeigneter fachlicher Semantik unterstützen zu können. Da wir davon ausgehen, dass gefundene Komponenten im Allgemeinen kontextspezifische Anpassungen (z. B. nach dem Adapter-Muster [GHJV01]) erfordern, werden weitere Eigenschaften einer Komponente wie z. B. die Signaturen ihrer Dienste im Rahmen der geschäftsprozessorientierten Komponentensuche nicht betrachtet.

Gegenstand der geschäftsprozessorientierten Komponentensuche sind aufgrund dieser starken Ausrichtung auf fachliche Aspekte serverseitige Fachkomponenten, wie sie in modernen betrieblichen Informationssystemen verbreitet Anwendung finden. Während Herzum und Sims in [HS00] der von ihnen identifizierten Klasse der Hilfskomponenten gute Aussichten auf marktmäßige Verbreitung einräumen, äußern sie Zweifel daran, dass Geschäftsobjektkomponenten als autonome Produkte auf einem Komponentenmarkt vermarktet werden können. Sie erwarten vielmehr, dass Geschäftsobjektkomponenten als Bestandteil größerer Frameworks eingesetzt werden, die eine bestimmte Menge betrieblicher Aktivitäten unterstützen. Solche Frameworks können beispielsweise durch (komplexe) Prozesskomponenten repräsentiert werden. In diesem Zusammenhang geben Herzum und Sims zu bedenken, dass Geschäftsprozesse hinsichtlich ihrer Funktionalität zwischen Branchen oft deutlich differieren und es daher schwierig sein kann, standardisierte Prozesskomponenten über Branchengrenzen hinweg zu nutzen. Allerdings halten wir es gerade für ein Merkmal der komponentenbasierten Softwareentwicklung, dass eine Vielzahl spezialisierter Anbieter die unterschiedlichen Branchen mit speziellen Lösungen abdecken. Bezüglich der Vermarktbarkeit von Geschäftsobjektkomponenten räumen Herzum und Sims ein, dass es erhebliche Standardisierungsbemühungen gibt. Wir gehen davon aus, dass diese Standardisierungsbemühungen die Vermarktbarkeit von Geschäftsobjektkomponenten zukünftig fördern werden. Im Rahmen der geschäftsprozessorientierten Komponentensuche betrachten wir daher sowohl Geschäftsobjekt- als auch Prozesskomponenten und konzentrieren uns auf deren Unternehmensschicht (vgl. Abschnitt 3.3.1).

5.2 Abstrakter Makroprozess geschäftsprozessorientierter Komponentensuche

Wie bereits in Abschnitt 3.2 eingeführt liegt dem Gedanken der Wiederverwendung von Komponenten eine Marktmetapher zugrunde, d. h. es wird davon ausgegangen, dass Komponenten auf einem elektronischen Marktplatz gehandelt werden. In unserem *abstrakten Makroprozess* geschäftsprozessorientierter Komponentensuche betrachten wir fünf Rollen von Marktteilnehmern:

- Ein *Marktplatzbetreiber* stellt die Infrastruktur für die Verwaltung von Komponenten bzw. deren Beschreibungen zur Verfügung. Dazu gehört insbesondere ein Komponenten-Repository. Neben Unternehmen, die auf den Vertrieb von Software spezialisiert sind, können beispielsweise Software entwickelnde Unternehmen wie SAP oder die DATEV, Technologieanbieter wie SUN oder Microsoft, aber auch (inter-)nationale Organisationen wie der Verband deutscher Maschinen- und Anlagenbau (VDMA) als Marktplatzbetreiber auftreten.

- Ein *Normierungsgremium* definiert fachliche und technische Standards, die die Interoperabilität von Komponenten fördern sowie die Suche nach Komponenten unterstützen. Zu diesen Standards gehören neben Schnittstellenspezifikationen (z. B. in Form von Signaturen oder XML-Dokumenttypen) insbesondere Sprachen für die fachliche Beschreibung von Komponenten und Anwendungsbereichen. Beispiele für Normierungsgremien sind die *Open Applications Group (OAG)* und die *Organization for the Advancement of Structured Information Standards (OASIS)*, die XML-Sprachen zur Unterstützung unternehmensübergreifender Geschäftsprozesse spezifizieren, sowie die DATEV, die gemeinsam mit dem FORWIN (Bayerischer Forschungsverbund Wirtschaftsinformatik) eine umfassende Funktionshierarchie zur Klassifizierung betriebswirtschaftlicher Software entwickelt hat.

- *(Komponenten-)Hersteller* entwickeln Komponenten, die u. U. komplex, d. h. aus anderen Komponenten zusammengesetzt sind, und bieten diese auf einem Marktplatz an. Bei der Entwicklung und Beschreibung von Komponenten berücksichtigen Komponentenhersteller Fachstandards, die durch Normierungsgremien vorgegeben werden.

- *(Komponenten-)Verwender* erwerben Komponenten auf dem Marktplatz und nutzen diese entweder im operativen Einsatz oder für die Entwicklung komplexer Komponenten. Bei der Komponentensuche berücksichtigen sie ggf. relevante Fachstandards.

- Ein *(Komponenten-)Broker* ist ein Dienstleister, der im Auftrag eines Verwenders auf dem Marktplatz verfügbare Komponenten sucht, die den Anforderungen des Verwenders möglichst genau entsprechen. Der Erwerb einer Komponente durch einen Verwender kann wahlweise durch den Broker in Verbindung mit dem Marktplatzbetreiber oder aber direkt mit deren Hersteller abgewickelt werden.

Es ist grundsätzlich vorstellbar, dass ein Marktteilnehmer mehrere Rollen gleichzeitig einnimmt. Ein Softwarehaus, das komplexe Komponenten durch Komposition von Komponenten anderer Hersteller entwickelt und anbietet, tritt beispielsweise als Hersteller und Verwender zugleich auf. Die geschäftsprozessorientierte Komponentensuche ist allerdings aufgrund ihrer Orientierung an Geschäftsprozessmodellen schwerpunktmäßig auf Endanwender bzw. Berater, also reine Verwender ausgerichtet, da diese ihre funktionalen Anforderungen – im Gegensatz zu Softwareentwicklern – verbreitet mit Hilfe von Geschäftsprozessmodellen spezifizieren.

Der in Abbildung 5.1 dargestellte Makroprozess der geschäftsprozessorientierten Komponentensuche sieht vier Phasen vor:

① *Angebotsphase*: Hersteller veröffentlichen Beschreibungen der von ihnen angebotenen Komponenten in einem Komponenten-Repository, das durch einen Marktplatzbetreiber zur Verfügung gestellt und von diesem gepflegt wird. Diese Beschreibungen umfassen insbesondere Informationen zu den von einer Komponente angebotenen Diensten sowie den kausal-logischen Einschränkungen ihrer Bereitstellung. Bei der fachlichen Beschreibung der Dienste berücksichtigen Komponentenhersteller Fachstandards, die durch ein Normierungsgremium formuliert werden.

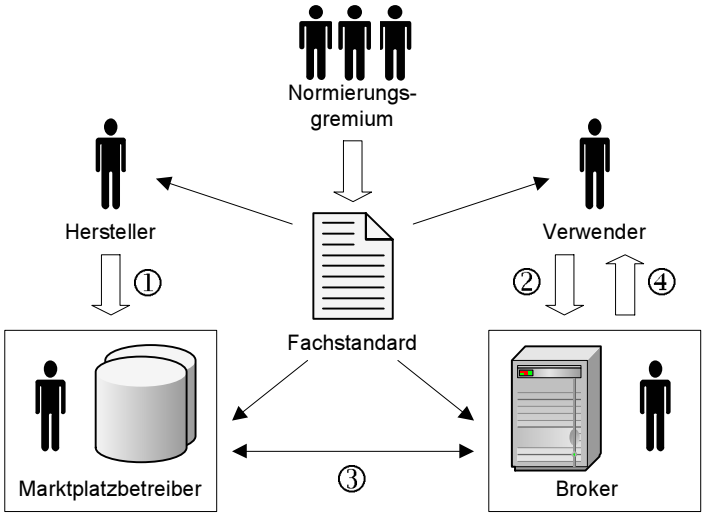


Abbildung 5.1: Makroprozess der geschäftsprozessorientierten Komponentensuche

② *Anforderungsphase*: Verwender, die am Erwerb von Komponenten interessiert sind, spezifizieren ihre Anforderungen an diese Komponenten gegenüber einem Broker in Form von Geschäftsprozessmodellen. Diese Geschäftsprozessmodelle beschreiben die betrieblichen Aktivitäten, die durch die gesuchte(n) Komponente(n) unterstützt werden sollen, sowie deren kausal-logische Ordnung. Die durchzuführenden betrieblichen Aktivitäten werden unter Bezugnahme auf einen Fachstandard spezifiziert.

③ *Suchphase*: Der Broker durchsucht die Komponenten-Repositories eines oder mehrerer Marktplatzbetreiber nach Komponenten, die die Ausführung der in einem Geschäftsprozessmodell spezifizierten Abläufe durch ihre Funktionalität geeignet unterstützen können. Dazu vergleicht er die im Geschäftsprozessmodell formulierten Anforderungen mit den Leistungen verfügbarer Komponenten hinsichtlich Eignung und Verfügbarkeit der von ihnen angebotenen Dienste.

④ *Präsentations- und Evaluationsphase*: Die durch den Broker gefundenen Kandidatenkomponenten werden dem Verwender präsentiert. Dabei kommen textuelle und graphische Darstellungsformen zum Einsatz. Der Verwender wird bei der Bewertung der Eignung gefundener Komponenten durch Metriken sowie die (graphische) Gegenüberstellung von Anforderungen und angebotenen Leistungen unterstützt.

Die inhaltliche Ausgestaltung der einzelnen Phasen durch die geschäftsprozessorientierte Komponentensuche wird in Abschnitt 5.4 konkreter beschrieben.

5.3 Komponentensuche als Subtyping-Problem

Komponentensuche setzt sich mit der Frage auseinander, ob eine Komponente in einem spezifischem Kontext (wieder-)verwendet werden kann, der zur Zeit ihrer Entwicklung

nicht in allen Details bekannt gewesen ist. Im objektorientierten Paradigma ist der Begriff der (Wieder-)Verwendbarkeit eng mit der Frage nach der *Substituierbarkeit* von Klassen bzw. ihren Objekten verknüpft. In beiden Fällen ist für eine gegebene Schnittstelle eine geeignete Implementierung zu finden. Während bei Wiederverwendungsproblemen im Allgemeinen noch keine solche Implementierung vorhanden bzw. bekannt ist, geht es bei der Frage der Substituierbarkeit darum, eine vorhandene Implementierung durch eine andere zu ersetzen. Dabei wird verlangt, dass der Austausch dieser Implementierung für einen Client der Schnittstelle nicht feststellbar ist. Der Begriff der Substituierbarkeit wird gewöhnlich durch das Konzept des *Subtyping* formal gefasst, d. h. durch die Feststellung von Subtyp-Beziehungen zwischen Klassen bzw. den durch sie implementierten Typen. Wegner und Zdonik beschreiben das *Prinzip der Substituierbarkeit* wie folgt:

> »*An instance of a subtype can always be used in any context in which an instance of a supertype was expected.*« [WZ88]

Ein *Typ* kann dabei als Sammlung von Objekten betrachtet werden, denen gewisse gemeinsame, extern beobachtbare Eigenschaften innewohnen [Ame91]. Beim *Subtyping* wird allgemein die Fragestellung betrachtet, ob ein Typ die Eigenschaften eines anderen Typs aufweist (die er durch weitere, nicht konkurrierende Eigenschaften ergänzen darf). Ist dies der Fall, so wird der erstgenannte Typ als *Subtyp* des zweiten Typs bezeichnet, der dann einen *Supertyp* des ersten Typs darstellt. Die Eigenschaften eines Supertyps können also als Anforderungen betrachtet werden, die ein Subtyp durch entsprechende Leistungen erfüllt. Eine solche Anforderung besteht z. B. darin, dass ein Subtyp alle Methoden des Supertyps anbietet (bzw. geeignete, z. B. durch ko- und kontravariante Veränderungen der Eingabe- und Ausgabeparametertypen erzeugte Varianten davon [Szy02]). Allerdings berücksichtigt dieser auf die Betrachtung von Signaturen ausgerichtete Begriff des Subtyping nicht das Verhalten eines Typs. So kann ein Typ eine Geschäftslogik definieren, die die zulässigen Sequenzen von Methodenaufrufen und damit die Wiederverwendbarkeit des Typs einschränkt. Daneben können zwei Methoden trotz gleicher Signatur ein abweichendes, im Extremfall sogar konträres Verhalten aufweisen. Da diese Einflüsse dazu führen können, dass eine bei bloßer Betrachtung von Signaturen bestehende Subtypeigenschaft bei zusätzlicher Berücksichtigung des Verhaltens verletzt wird, ergänzen das *verhaltensorientierte* bzw. das *zustandsbasierte Behavioural Subtyping* [Weh02] den Vergleich von Signaturen um die zusätzliche Betrachtung des Verhaltens von Typen.

Übertragen auf das Problem der geschäftsprozessorientierten Komponentensuche kann ein Geschäftsprozess vergleichbar mit einem Supertyp als eine Zusammenfassung verhaltensorientierter Anforderungen angesehen werden. Ziel der geschäftsprozessorientierten Komponentensuche ist es nun, Komponenten zu finden, die vergleichbar mit einem Subtyp zumindest Teile dieser Anforderungen durch ihre Leistungen erfüllen. Grob gesagt erfüllt eine Komponente die Anforderungen eines Geschäftsprozesses, wenn sie zum einen Dienste anbietet, die die Ausführung der einzelnen Aktivitäten des Geschäftsprozesses geeignet unterstützen, und zum anderen eine Geschäftslogik definiert, die die Nutzung dieser Dienste in der durch den Geschäftsprozess vorgegebenen Abfolge gestattet. Auf einer höheren Abstraktionsebene beschreiben Geschäftsprozessmodelle Ablaufvarianten der in einem Unternehmen auszuführenden Geschäftsprozesse, während Komponentenbeschreibungen die von Komponenten angebotene Funktionalität spezifizieren, die für

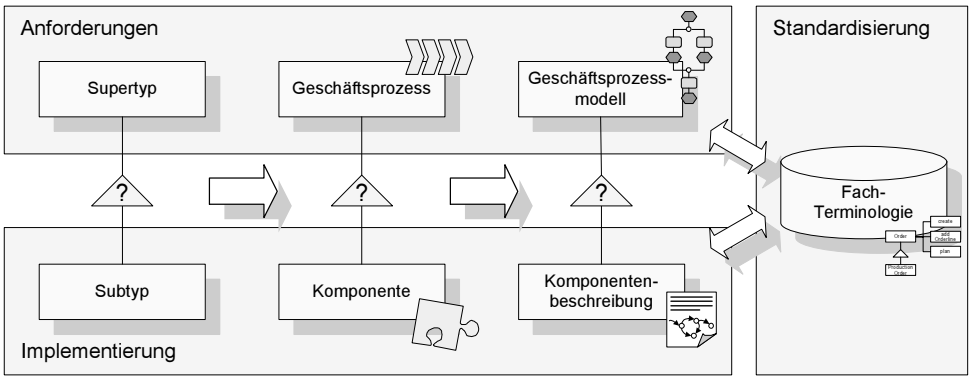


Abbildung 5.2: Geschäftsprozessorientierte Komponentensuche als Subtyping-Problem

die Unterstützung der Ausführung von Geschäftsprozessen zur Verfügung steht. Um die fachliche Zuordnung von Diensten einer Komponente zu betrieblichen Aktivitäten eines Geschäftsprozesses zu ermöglichen, sehen wir die Verwendung einer standardisierten Fachterminologie für die Spezifikation der fachlichen Semantik von Aktivitäten und Diensten vor. Abbildung 5.2 fasst unsere Interpretation der geschäftsprozessorientierten Komponentensuche als Subtyping-Problem graphisch zusammen.

5.4 Konkreter Makroprozess geschäftsprozessorientierter Komponentensuche

Wir stellen nun die konzeptionelle Umsetzung der geschäftsprozessorientierten Komponentensuche als Konkretisierung der einzelnen Phasen des abstrakten *Makroprozesses* aus Abschnitt 5.2 vor:

① *Angebotsphase*: Für die Beschreibung von Komponenten, die auf einem Marktplatz angeboten werden sollen, verwenden wir mit *Component Description Language (CDL)* eine Sprache zur integrierten Darstellung struktureller und dynamischer Aspekte von Komponenten. Da Komponenten oft eine Geschäftslogik implementieren, die die zulässigen Sequenzen der Nutzung ihrer Dienste einschränkt, unterstützt CDL die explizite Spezifikation der Verfügbarkeit von Diensten in Form so genannter Protokollmaschinen. Abschnitt 6.1 führt die Component Description Language ein.

② *Anforderungsphase*: Bei der Spezifikation von Anforderungen in Form von Geschäftsprozessmodellen konzentrieren wir uns im Rahmen dieser Arbeit auf ablauforganisatorische Aspekte, während Anforderungen an die Aufbauorganisation oder den Datenfluss etc. unberücksichtigt bleiben. Für die Formulierung von Geschäftsprozessmodellen setzen wir lineare Prozessmodelle ein. Abschnitt 6.2 begründet die Entscheidung zugunsten der linearen Prozessmodellierung.

Interoperabilitätsebene	**Subtyping-Konzept**	**Ansatz**
Signaturebene	Subtyping (Signature Matching)	—
Protokollebene	Behavioural Subtyping (verhaltensorientiert)	• Repräsentation des Verhaltens von Geschäftsprozessen und Komponenten mittels beschrifteter Transitionssysteme • Subtyping-Relation auf Transitionssystemen
Semantikebene	Behavioural Subtyping (zustandsbasiert)	• Definition der Semantik von Geschäftsprozessen und Komponenten mittels normsprachlicher Aussagen • Spezialisierungsrelation auf normsprachlichen Aussagen

Tabelle 5.1: Interoperabilitätsebenen, Subtyping-Konzepte und verfolgter Ansatz

③ *Suchphase*: Gegenstand der Suchphase ist die Bestimmung von Komponenten zur Unterstützung der Ausführung von Geschäftsprozessen, indem wie im vorangegangenen Abschnitt skizziert ein Subtyping-Ansatz verfolgt wird. Überträgt man das Prinzip der Substituierbarkeit nach Wegner und Zdonik auf die geschäftsprozessorientierte Komponentensuche, so soll eine geeignete Komponente (der Subtyp) überall dort *benutzt* werden können, wo ein Geschäftsprozess (der Supertyp) auszuführen ist. Nach [VHT99] können drei Ebenen der Benutzbarkeit oder Interoperabilität unterschieden werden: *Signaturebene* (Bezeichnung sowie Argument- und Rückgabetypen von Diensten), *Protokollebene* (partielle Ordnung zwischen Diensten) und *Semantikebene* (die „Bedeutung“ der Dienste). Tabelle 5.1 ordnet diesen Interoperabilitätsebenen die ihnen entsprechenden Subtyping-Konzepte zu. Sie stellt darüber hinaus stichwortartig dar, welchen Ansatz wir zur Lösung des Problems der Komponentensuche auf den einzelnen Ebenen verfolgen.

Im Rahmen dieser Arbeit konzentrieren wir uns auf die Protokoll- und die Semantikebene. Vergleiche auf der Signaturebene als Ansatz zur Suche wiederverwendbarer Komponenten werden in [ZW93] vorgestellt (vgl. auch Abschnitt 4.3.4). Zwar ließe sich auch für Aktivitäten eines Geschäftsprozessmodells eine Signatur ermitteln, die mit der Signatur eines Dienstes verglichen werden könnte, allerdings erscheint der syntaktische Vergleich der Bezeichnungen von geforderten Aktivitäten und angebotenen Diensten wenig erfolgversprechend.

Für den Vergleich von Geschäftsprozessmodellen und Komponentenbeschreibungen auf der Protokollebene greifen wir direkt auf den Ansatz des verhaltensorientierten Behavioural Subtyping zurück und untersuchen eine Subtyping-Relation zwischen beschrifteten Transitionssystemen, mit denen wir die Dynamik von Geschäftspro-

zessmodellen und Komponentenbeschreibungen formal repräsentieren. Dieser Protokollvergleich wird in Kapitel 7 vorgestellt.

In Anlehnung an das zustandsbasierte Behavioural Subtyping gründet der Vergleich von Geschäftsprozessmodellen und Komponentenbeschreibungen auf der Semantikebene auf der Idee, Aktivitäten und Diensten eine fachliche Semantik zuzuordnen, die dann über eine Spezialisierungsrelation miteinander verglichen werden kann. Für die Spezifikation dieser fachlichen Semantik verfolgen wir einen normsprachlichen Ansatz. Kapitel 8 präsentiert unsere Konzepte für den semantischen Vergleich von Geschäftsprozessmodellen und Komponentenbeschreibungen.

④ *Präsentations- und Evaluationsphase*: Die Präsentation gefundener Komponenten mittels textueller und graphischer Darstellungsformen sowie die Unterstützung ihrer Bewertung durch Metriken und die Gegenüberstellung von Anforderungen und angebotenen Leistungen stellt keinen Schwerpunkt dieser Arbeit dar und wird daher im Folgenden nicht untersucht.

Abbildung 5.3 stellt der Angebots-, Anforderungs- und Suchphase noch einmal die ihr zugeordnete Aufgabe graphisch gegenüber und verweist auf das Kapitel, in denen diese Aufgabe konzeptionell bearbeitet wird. Diese Abbildung wird in den folgenden Kapiteln als Wegweiser durch den zweiten Teil dieser Arbeit dienen.

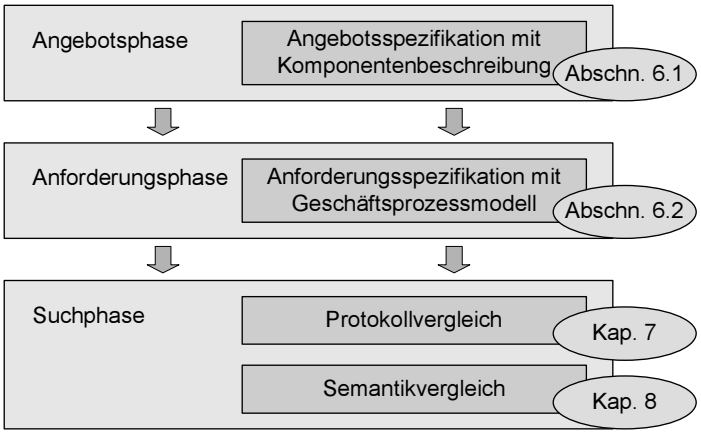


Abbildung 5.3: Übersicht über Phasen sowie zugeordnete Aufgaben und Kapitel

5.5 Zusammenfassung

In diesem Kapitel haben wir einen Überblick über den Ansatz der geschäftsprozessorientierten Komponentensuche gegeben. Dabei haben wir zunächst die Verwendung von Geschäftsprozessmodellen zur Anfragespezifikation, die Bereitstellung einer umfassenden Komponentenbeschreibungssprache, die sich gleichermaßen an Technologie- und Fachexperten richtet, sowie die Konzeption eines Suchverfahrens, das insbesondere das in Suchanfragen festgehaltene Prozesswissen bei der Komponentensuche berücksichtigt, als

zentrale Ziele der geschäftsprozessorientierten Suche nach serverseitigen Geschäftsobjekt- und Prozesskomponenten definiert. Im Rahmen der Vorstellung des Makroprozesses der geschäftsprozessorientierten Komponentensuche haben wir fünf Rollen von Akteuren eingeführt, die in vier Phasen von der Angebots- und Anfrageformulierung über die eigentliche Komponentensuche bis zur Präsentation der Suchergebnisse auf einem Komponentenmarkt miteinander interagieren. Anschließend haben wir unsere Interpretation der geschäftsprozessorientierten Komponentensuche als Subtyping-Problem vorgestellt und die Übertragung wesentlicher Gedanken des Behavioural Subtyping auf die Komponentensuche motiviert. Schließlich haben wir die konkrete Ausgestaltung der Phasen zur Angebots- und Anfrageformulierung sowie der Komponentensuche im Rahmen unserer Arbeit skizziert.

Kapitel 6

Prozessorientierte Beschreibung von Komponenten und Anforderungen

Die Wiederverwendbarkeit von Komponenten im Rahmen der komponentenbasierten Softwareentwicklung setzt die Verfügbarkeit einer umfassenden Dokumentation ihrer Struktur und ihres abstrakten Verhaltens voraus. In der Angebotsphase des Makroprozesses geschäftsprozessorientierter Komponentensuche veröffentlichen Komponentenhersteller entsprechende Komponentenbeschreibungen, um ihre Komponenten auf einem Komponentenmarkt anzubieten. Verwender spezifizieren ihre Anforderungen an geeignete Komponenten in der anschließenden Anforderungsphase in Form von Geschäftsprozessmodellen. Abbildung 6.1 stellt die Einordnung dieser Aktivitäten in den Makroprozess der geschäftsprozessorientierten Komponentensuche graphisch dar.

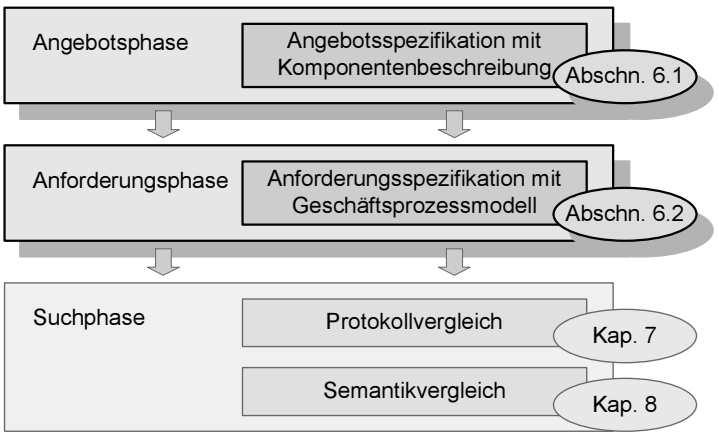


Abbildung 6.1: Einordnung des Kapitels in den Makroprozess

Gegenstand dieses Kapitels ist die Einführung bzw. Auswahl von Sprachen für die prozessorientierte Beschreibung der Leistungen angebotener Komponenten sowie der Anforderungen an gesuchte Komponenten. Für erstgenannte Aufgabe stellen wir in Abschnitt 6.1 die Komponentenbeschreibungssprache *Component Description Language* vor,

letztgenannte Aufgabe behandeln wir in Abschnitt 6.2 durch die Verwendung linearer Prozessmodelle. Wir beschließen das Kapitel in Abschnitt 6.3 mit einer Zusammenfassung.

6.1 Komponentenbeschreibungen

Ziel dieses Abschnitts ist die Vorstellung der Komponentenbeschreibungssprache *Component Description Language (CDL)*[1], die die Beschreibung von Fachkomponenten aus einer fachlich orientierten und damit für Fachexperten verständlicheren Sichtweise gestattet. Ausgehend von einer groben Charakterisierung der CDL erläutern wir zunächst das ihr zugrunde liegende Komponentenmodell und führen anschließend ihre Syntax und Semantik informell ein.

6.1.1 Charakterisierung

Bei der komponentenbasierten Softwareentwicklung durch Wiederverwendung werden Komponenten im Allgemeinen als *Black Boxes* betrachtet, d. h. die Implementierungsdetails einer Komponente bleiben dem Wiederverwender verborgen [Szy02]. Da sich diese *Black-Box-Wiederverwendung* allein auf Komponentenbeschreibungen abstützt, fällt diesen die wichtige Aufgabe zu, die korrekte Verwendung der von einer Komponente angebotenen Schnittstellen auf technischer und fachlicher Ebene sowie ihre Kontextabhängigkeiten umfassend zu dokumentieren. Shaw und Garlan formulieren in [SG96] Anforderungen an Architekturbeschreibungssprachen, die sich wie folgt auf Komponentenbeschreibungssprachen übertragen lassen:

- *Composition (Komposition)*: Eine Komponentenbeschreibungssprache sollte es ermöglichen, ein Softwaresystem als eine Komposition unabhängiger Komponenten zu beschreiben.
- *Abstraction (Abstraktion)*: Die Beschreibung von Komponenten und ihren Interaktionen sollte so klar und präzise erfolgen können, dass von einem Wiederverwender auf ihre Einsatzmöglichkeit im Rahmen einer Komposition geschlossen werden kann.
- *Reusability (Wiederverwendbarkeit)*: Die Wiederverwendung von Komponenten, aber auch von Systemfamilien, die durch Architekturmuster im Sinne eines Frameworks definiert sind, sollte durch eine Beschreibungssprache unterstützt werden.
- *Configuration (Konfiguration)*: Die Beschreibung der Struktur eines komponentenbasierten Softwaresystems sollte unabhängig von den Beschreibungen der strukturierten Komponenten erfolgen können.
- *Heterogeneity (Heterogenität)*: Eine Komponentenbeschreibungssprache sollte die Möglichkeit bieten, heterogene Beschreibungen von Komponenten oder Architekturmustern in einem System zu kombinieren.

[1] Die *Business Object Component Architecture (BOCA)* [OMG98] verwendet für ihre *Component Definition Language* dieselbe Abkürzung. Die Component Description Language stellt eine davon unabhängige Komponentenbeschreibungssprache dar.

- *Analysis (Analyse)*: Die Durchführung verschiedenartiger Analysen des beschriebenen Softwaresystems sollte durch eine Beschreibungssprache unterstützt werden.

Mit Ausnahme der Heterogenitätsanforderung, die in CDL keine explizite Unterstützung findet, berücksichtigt CDL alle diese Anforderungen. Die Beschreibung von Softwaresystemen als Kompositionen unabhängiger Komponenten ermöglicht CDL dadurch, dass Komponenten und Kontrakte als zentrale Elementtypen im CDL-Komponentenmodell Berücksichtigung finden. Die Wiederverwendbarkeit einzelner Komponenten wird dadurch unterstützt, dass Komponenten durch eigenständige Beschreibungen dokumentiert werden. Architekturmuster, die z. B. zum Aufbau von Systemfamilien wiederverwendet werden sollen, werden in CDL durch die explizite Repräsentation von Komponenten berücksichtigt, die im Sinne eines Komponentenframeworks erweitert werden müssen. Der Aufbau von Kompositionsstrukturen durch die Konfiguration (Parametrisierung) solcher Komponenten mit anderen Komponenten wird in CDL durch das Konstrukt der komplexen Komponente explizit unterstützt, das Verweise auf die strukturierten Komponenten verwaltet.

Das Problem, Komponenten auf geeignetem Abstraktionsniveau beschreiben zu können, ist bereits in vielen Ansätzen angegangen worden (vgl. Abschnitt 3.4). Die Entwicklung der CDL wurde durch die Anforderung getrieben, fachlich orientierte Beschreibungen bei hinreichend formaler Fundierung zu ermöglichen. Einerseits sollten CDL-Komponentenbeschreibungen nicht nur für Technologieexperten, sondern auch für Fachexperten verständlich sein, d. h. neben technischen Aspekten wie Schnittstellen und Struktur einer (komplexen) Komponente sollten auch fachliche Aspekte wie z. B. die Semantik von Diensten oder der „Lebenszyklus“ von Geschäftsobjekten in einer Komponentenbeschreibung erfasst werden können. In dieser Hinsicht verfolgt CDL den Gedanken, den Ritter in [Rit98] angeregt und in seine Beschreibungssprache *Simple Component Description Language (SCDL)* [Rit00] einfließen lassen hat: Ähnlich wie große, betriebliche Standardsoftwareprodukte wie SAP R/3 durch strukturierte, semi-formale Beschreibungen (so genannte *Softwarereferenzmodelle* [WWB+99, AR00]) dokumentiert werden, sollen Komponenten und deren Kompositionen durch „komponentenorientierte Softwarereferenzmodelle“ beschrieben werden. Andererseits sollte CDL – anders als einige Sprachen zur Erstellung von Softwarereferenzmodellen (vgl. die Kritik in [Rum99]) – eine hinreichend formale Grundlage aufweisen, um die im Rahmen der geschäftsprozessorientierten Komponentensuche erforderlichen Analysen zu ermöglichen.

Wie aus den Überlegungen zur geschäftsprozessorientierten Komponentensuche in Kapitel 5 folgt, konzentriert sich CDL auf die Beschreibung der auf der Unternehmensschicht angesiedelten Geschäftslogik von Fachkomponenten. Weitere, durchaus relevante Informationen wie z. B. zu Hersteller, erforderlicher Plattform, Quality of Service, Performance oder Lizenzierung werden bei der geschäftsprozessorientierten Komponentensuche nicht berücksichtigt und sind daher nicht Gegenstand der CDL. Abbildung 6.2 stellt den im Memorandum zur *Vereinheitlichten Spezifikation von Fachkomponenten* (vgl. Abschnitt 3.4) identifizierten Beschreibungsebenen die Berücksichtigung durch CDL-Konzepte gegenüber. In diesem Abschnitt konzentrieren wir uns auf die Einführung der formalen Konzepte zu Beschreibung von Komponenten auf der Schnittstellen-, Abstimmungs- und Verhaltensebene. Die Konzepte zur Herstellung des fachlichen Bezugs einer Komponente auf der Terminologie- und Aufgabenebene ergänzen wir in Kapitel 8.

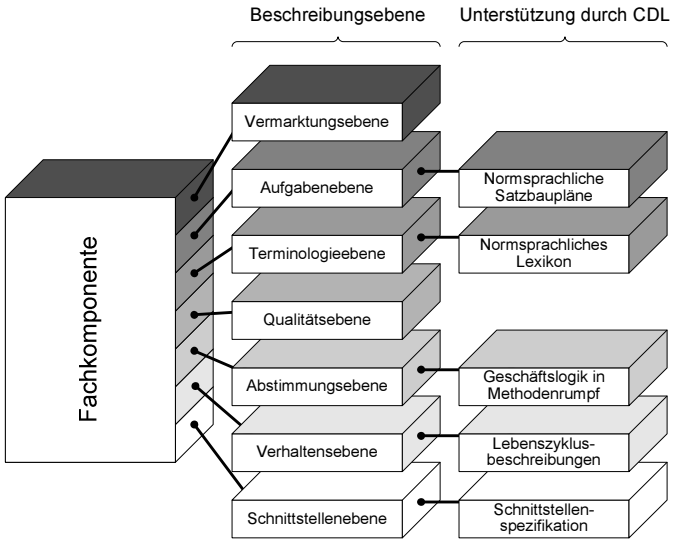


Abbildung 6.2: Berücksichtigung der Beschreibungsebenen durch CDL

6.1.2 Komponentenmodell

Das Komponentenmodell der CDL stellt eine einheitliche Abstraktion von den Komponentenmodellen der in Abschnitt 3.3 vorgestellten Komponententechnologien COM, EJB und CCM dar (vgl. auch [TR00]). Andere Komponentenmodelle wie z. B. das Modulkonzept betriebswirtschaftlicher Standardsoftwareprodukte wie SAP R/3 werden durch CDL nicht explizit berücksichtigt. Da CDL eine Weiterentwicklung der SCDL darstellt, ergeben sich fast zwangsläufig Parallelen zwischen dem Komponentenmodell der SCDL und dem der CDL. Letzteres ist in Abbildung 6.3 in Form eines Klassendiagramms graphisch dargestellt.

Bislang hatten wir in unserer Darstellung der Komponentensoftware zwischen Geschäftsobjekt- und Prozesskomponenten unterschieden. Diese Differenzierung findet sich im CDL-Komponentenmodell nicht wieder, da wir beide Arten von Komponenten im Kontext der geschäftsprozessorientierten Komponentensuche gleich behandeln. Zentrales Element des CDL-Komponentenmodells ist der abstrakte Begriff der *Komponente*, von der sich die spezielleren Varianten *atomare Komponente* und *komplexe Komponente* im Sinne des *Kompositum*-Entwurfsmusters [GHJV01] ableiten. Atomare Komponenten können eine *direkte Schnittstelle* und – über die möglicherweise von ihr realisierten *Klassen* – *indirekte Schnittstellen* anbieten, da sowohl atomare Komponenten als auch Klassen *Klassifizierer* darstellen und somit eine Menge von *Operationen* als Schnittstelle exportieren. Instanzen einer Klasse (die *Geschäftsobjekte*) werden über die direkte Schnittstelle einer atomaren Komponente erzeugt und lassen sich über ihre eigene Schnittstelle (die Teil der indirekten Schnittstelle der atomaren Komponente ist) ansprechen. Aufgrund der Unterstützung von direkten und indirekten Schnittstellen gestattet das CDL-Komponentenmodell die Repräsentation von Komponenten des typbasierten und des instanzbasierten Architekturstils (vgl. Abschnitt 3.1).

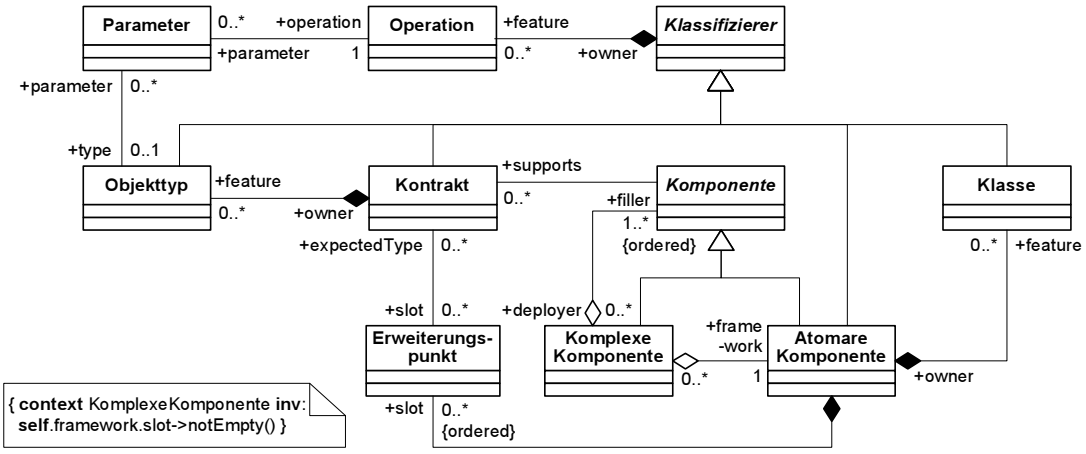


Abbildung 6.3: CDL-Komponentenmodell

Atomare Komponenten können ähnlich wie Komponentenframeworks *Erweiterungspunkte* definieren, über die sie im Sinne einer Komposition mit anderen Komponenten verbunden werden können. Zur begrifflichen Unterscheidung bezeichnen wir atomare Komponenten, die solche Erweiterungspunkte anbieten, als *offene (atomare) Komponenten*, während wir atomare Komponenten ohne Erweiterungspunkte *geschlossene (atomare) Komponenten* nennen. Offene Komponenten stellen damit die Basis für die Erstellung komplexer Komponenten dar, die aus der Komposition mehrerer Komponenten hervorgehen. Eine komplexe Komponente besteht aus einer offenen Komponente, deren Erweiterungspunkte mit geeigneten (atomaren oder wiederum komplexen) Komponenten parametrisiert sind. Da auf jeder Stufe einer derartigen Kompositionshierarchie der gleiche Komponentenbegriff anzutreffen ist, kann das CDL-Komponentenmodell als kontinuierlich rekursiv (im Gegensatz zu diskret rekursiv) bezeichnet werden. Anders als eine atomare Komponente definiert eine komplexe Komponente keine eigene (direkte und/oder indirekte) Schnittstelle, sondern exportiert die Schnittstellen der verwendeten offenen Komponente.

Die Parametrisierung der Erweiterungspunkte einer offenen Komponente mit Komponenten wird durch deren Typisierung mit *Kontrakten* kontrolliert. Kontrakte repräsentieren ein zu den Komponenten auf Seiten der Implementierung analoges Konzept auf der Typebene: Ähnlich wie atomare Komponenten eine direkte Schnittstelle sowie (über Klassen) indirekte Schnittstellen definieren, spezifizieren Kontrakte neben einer direkten Schnittstelle mittels so genannter *Objekttypen* indirekte Schnittstellen. Objekttypen repräsentieren dabei Typen von Geschäftsobjekten, die durch Klassen einer atomaren Komponente realisiert und neben Basisdatentypen wie `int`, `float` und `String` als Typen der *Parameter* von Operationen verwendet werden können. Eine Komponente stellt eine gültige Parametrisierung eines Erweiterungspunkts dar, wenn sie den vom Erweiterungspunkt vorgegebenen Kontrakt unterstützt, d. h. wenn sie mindestens die im Kontrakt geforderte direkte Schnittstelle anbietet und darüber hinaus noch Klassen realisiert, die die Objekttypen des Kontrakts implementieren. Dabei kann eine Klasse mehrere Objekttypen

gleichzeitig implementieren. Tatsächlich sind die Anforderungen an eine Komponente für die Unterstützung eines Kontrakts komplexer, da CDL neben den hier vorgestellten strukturellen Eigenschaften auch Verhaltensaspekte berücksichtigt. So erfordert die Analyse der Unterstützung eines Kontrakts durch eine Komponente vollständige Spezifikationen des im Kontrakt geforderten sowie des von der Komponente gezeigten Verhaltens. Da wir uns hier mit Unterspezifikationen des Verhaltens beschäftigen, ist die genaue Definition der Erfüllung eines Kontrakts nicht Gegenstand dieser Arbeit. Die bereits aus Abschnitt 3.3 bekannte Abgrenzung von Typ-, Implementierungs- und Laufzeitsicht auf eine Komponente ist in Abbildung 6.4 noch einmal für die Konzepte des CDL-Komponentenmodells dargestellt.

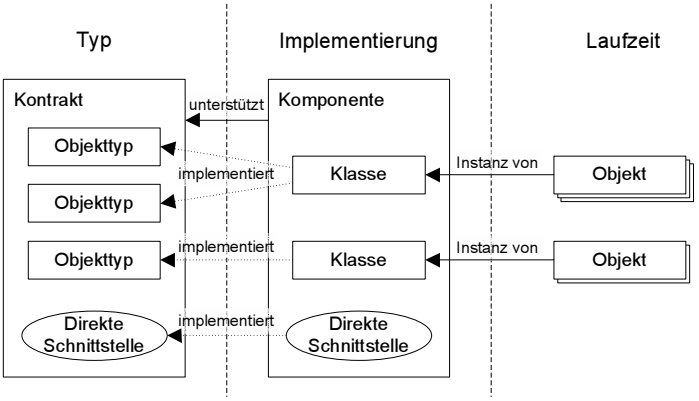


Abbildung 6.4: Komponentenmodell der Component Description Language (CDL)

Tabelle 6.1 stellt die zentralen Konstrukte des CDL-Komponentenmodells für die Repräsentation von Kontrakten und geschlossenen atomaren Komponenten den entsprechenden Konstrukten der Komponentenmodelle von COM, EJB und CCM gegenüber. Die hier nicht aufgeführten CDL-Konstrukte für offene atomare Komponenten und komplexe Komponenten bieten eine geeignete Abstraktion für die Darstellung von Abhängigkeiten zwischen Komponenten der jeweiligen Komponentenmodelle. Der in COM mit dem Konzept der Kategorien verfolgte Ansatz zur Spezifikation von Kontrakten höherer Ordnung (vgl. Abschnitt 3.3.2) wird im CDL-Komponentenmodell von der Ebene der Klassen auf die Komponentenebene verschoben, d. h. ein Kontrakt wird durch eine Komponente statt durch einzelne Klassen unterstützt. Diese Verschiebung schließt allerdings die Definition von Kontrakten für einzelne COM-Klassen nicht aus, da COM-Klassen als Komponenten mit einer Klasse und COM-Kategorien als Kontrakte mit je einem Objekttyp pro COM-Schnittstelle beschrieben werden können.

6.1.3 Syntax und Semantik

Sowohl Geschäftsobjekt- als auch Prozesskomponenten zeichnen sich dadurch aus, dass die von ihnen implementierte Geschäftslogik einen starken Bezug zu den Geschäftsprozessen aufweisen, die sie unterstützen sollen [HS00]. So implementieren Geschäftsobjektkomponenten oft einen spezifischen Lebenszyklus der von ihnen bereitgestellten Geschäftsob-

Kriterium	**COM**	**EJB**	**CCM**	**CDL**
Kontrakt	Kategorie	—	Component Type	Kontrakt
Indirekte Schnittstelle	Schnittstelle	Component Interface	Facet	Objekttyp
Direkte Schnittstelle	IClassFactory (COM API)	Home Interface	Home Interface	Operationen des Kontrakts
Komponente	COM-Komponente	Enterprise Bean	Component	Komponente
Indirekte Schnittstelle	COM-Klasse	Enterprise Bean Class + EJBObject	Facets Implementierung (nicht spez.)	Klasse
Direkte Schnittstelle	ClassFactory (+ COM-Bibliothek)	Enterprise Bean Class + EJBHome	Home Interface Implementierung	Operationen der Komponente

Tabelle 6.1: Gegenüberstellung der Komponentenmodelle von COM, EJB, CCM und CDL

jekte (ein Auftrag muss z. B. freigegeben werden, bevor er ausgeführt werden kann). Prozesskomponenten fassen Dienste für die Ausführung eines bestimmten Geschäftsprozesses zusammen, zwischen denen – ähnlich wie bei Geschäftsobjektkomponenten – bestimmte Reihenfolgebeziehungen vorgegeben sein können (nach Auftragseingang erfolgt beispielsweise zunächst eine Bonitätsprüfung, bevor der Auftrag angenommen oder abgelehnt wird). Aufgrund dieses engen Zusammenhangs zwischen Fachkomponenten und den von ihnen unterstützten Geschäftsprozessen stellte die Repräsentation der Geschäftslogik von Fachkomponenten bzw. ihren Geschäftsobjekten neben der Beschreibung struktureller Aspekte eine wesentliche Anforderung beim Entwurf der CDL dar.

Laut Harel und Gery umfasst eine minimale Menge von Sprachen zur Beschreibung objektorientierter Systeme *Klassendiagramme* für die Darstellung statischer Strukturen sowie *Statecharts* zur Spezifikation der Dynamik des Systems [HG97]. Insbesondere hinsichtlich der Beschreibung von Verhalten existieren jedoch einige interessante Alternativen zu Statecharts bzw. deren in der *Unified Modeling Language (UML)* [OMG03] verwendeten Variante der *Zustandsmaschinen*, die wir bereits in Abschnitt 3.4 vorgestellt und diskutiert haben. Da ein Anspruch an die CDL darin besteht, eine möglichst hohe Verständlichkeit gerade für Fachexperten zu erzielen, haben sich operationelle Repräsentationsformen für die Darstellung des Verhaltens von Komponenten und deren Geschäftsobjekten angeboten. Anders als bei ihrem Vorgänger SCDL, deren Semantik auf der Grundlage des π-Kalküls definiert ist, basiert die Beschreibung der Dynamik in CDL auf so genannten *Protokollmaschinen* (*Protocol State Machines*) [OMG03], einer Variante der UML-Zustandsmaschinen für die Spezifikation der Lebenszyklen von Objekten. Von SCDL bzw. dem π-Kalkül wurde diesbezüglich neben dem grundsätzlichen Ziel, operationelle Beschreibungen von Lebenszyklen zu ermöglichen, lediglich die Syntax zur Darstellung von empfangenen und gesendeten Operationsaufrufen übernommen.

Für die Beschreibung struktureller Eigenschaften greift CDL – genau wie SCDL – auf grundlegende Konzepte von Klassendiagrammen zurück.

Wie bereits aus dem CDL-Komponentenmodell hervorgeht, unterscheidet *CDL* zwischen Komponenten- und Kontraktbeschreibungen, die sich wie folgt charakterisieren lassen. Komponentenbeschreibungen definieren eine obere Grenze der Menge von Interaktionssequenzen, an der die Komponente (evtl. mittels ihrer Geschäftsobjekte) teilnehmen kann. Diese Semantik ist aus Sicherheitsbetrachtungen heraus interessant, wobei der Begriff der *Sicherheit (Safety)* nach Owicki und Lamport besagt, dass „nie etwas Schlechtes passiert“ [OL82]: Eine Komponente geht nicht mehr Interaktionen ein, als durch ihre Beschreibung zugesichert wird. Demgegenüber spezifizieren Kontraktbeschreibungen eine untere Grenze der Menge erwarteter Interaktionssequenzen und damit die geforderte *Lebendigkeit (Liveness)* einer Komponente. Dabei umschreibt der Begriff der Lebendigkeit vereinfachend ausgedrückt, dass „schließlich etwas Gutes passiert“ [OL82], also erwartete Interaktionen stattfinden. Diese Semantik stellt sicher, dass eine Komponente, die einen bestimmten Kontrakt unterstützt, auch mindestens die erforderlichen Dienste anbietet. Ebert und Engels haben die Begriffe *beobachtbares Verhalten* (*observable behaviour*) und *aufrufbares Verhalten* (*invocable behaviour*) zur Benennung dieser verschiedenen Semantiken von Verhaltensbeschreibungen geprägt [EE94]. Analoge Formulierungen finden sich auch auf dem Gebiet der Prozessalgebren: Ein Prozess *kann* bzw. *muss* an einer Menge von Traces teilnehmen („may engage“ bzw. „must engage“) [Old91]. Da CDL jedoch explizit Unterspezifikationen von Komponenten und Kontrakten unterstützt (z. B. durch die Möglichkeit, nichtdeterministische Entscheidungen zu spezifizieren), darf die vorangegangene Charakterisierung von Komponenten- und Kontraktbeschreibungen nicht strikt (z. B. im Sinne einer Garantie des Komponentenherstellers) interpretiert werden. Sie stellt stattdessen lediglich eine Art Richtschnur dar, an der sich „gute“ Beschreibungen orientieren.

In den folgenden Abschnitten führen wir die einzelnen Konstrukte der CDL anhand einfacher Beispielkomponenten ein (vgl. auch [TR00]). Wir stellen dazu zunächst eine Beschreibung einer geschlossenen atomaren Komponente vor und gehen kurz auf die Beschreibung von Kontrakten ein. Anschließend spezifizieren wir eine offene atomare Komponente, auf deren Grundlage wir schließlich eine einfache komplexe Komponente konstruieren. Dabei verzichten wir auf die genaue Erläuterung einzelner, syntaktischer Details zugunsten einer übersichtlicheren Darstellung und verweisen auf Anhang A, in dem die vollständige Grammatik der CDL angegeben ist.

Geschlossene atomare Komponenten In diesem Abschnitt führen wir die syntaktischen Konstrukte der CDL ein, die für die Beschreibung geschlossener atomarer Komponenten benötigt werden. Als illustrierendes Beispiel verwenden wir eine Komponente namens `AccountComponent`, die einfache Dienste für die Verwaltung und Benutzung von Konten anbietet (siehe Listing 6.1).

Komponentenbeschreibungen werden durch das Schlüsselwort `component` eingeleitet. Sie können Beschreibungen einer beliebigen Anzahl von Klassen enthalten, die durch das Schlüsselwort `class` gekennzeichnet sind und die von der Komponente implementierten Typen von Geschäftsobjekten spezifizieren. Die Komponente `AccountComponent` in Listing 6.1 beinhaltet nur eine Klasse namens `Account`. Die direkte Schnittstelle einer

```
component AccountComponent {
    class Account {
        deposit(in amount: float) {
            return;     /* Erhöhung des Kontostands nicht spezifiziert */
        };
        withdraw(in amount: float) {
            return;     /* Reduzierung des Kontostands nicht spezifiziert */
        };
        close() {
            return;     /* Schließen des Kontos nicht spezifiziert */
        };

        created()              =   initialized();
        initialized()          =   deposit?(amount).balance(amount)
                                 + close?().closed();
        balance(value: float) =   deposit?(amount).balance(value+amount)
                                 + on withdraw?(amount)
                                     if (amount <= value)
                                       do balance(value-amount)
                                 + on close?()
                                     if (value == 0)
                                       do closed();
        closed()               =   ();
    };

    openAccount(out account: AccountComponent.Account) {
        new(AccountComponent.Account account);
        return(account);
    };

    created()            = initialized();
    initialized()        = openAccount?(account).initialized();
};
```

Listing 6.1: Geschlossene atomare Komponente `AccountComponent`

Komponente besteht aus den Operationen, die von der Komponente direkt, d. h. nicht über ihre Geschäftsobjekte, angeboten werden. Aufgrund des prozeduralen Charakters der direkten Schnittstelle bezeichnen wir die enthaltenen Operationen auch als *Prozeduren*. In unserem Beispiel besteht die direkte Schnittstelle von `AccountComponent` aus einer Prozedur `openAccount()`, mit der ein neues Konto angelegt werden kann. Die indirekte Schnittstelle einer Komponente besteht aus ihren Klassen und den von ihnen angebotenen Operationen. Die Operationen der indirekten Schnittstelle bezeichnen wir wegen ihres objektorientierten Charakters auch als *Methoden*.[2] Die Klasse `Account` definiert Methoden `deposit()` und `withdraw()` zum Ein- und Auszahlen von Geldbeträgen sowie `close()` zum Schließen des Kontos. Parameter von Operationen sind gerichtet und gemäß CDL-Komponentenmodell mit Objekttypen sowie Basisdatentypen wie `int`, `float` und `String` typisiert. Aus praktischen Erwägungen lassen wir in CDL darüber hinaus

[2]Sofern die Unterscheidung zwischen Prozeduren und Methoden für die Erläuterungen nicht relevant ist, sprechen wir weiterhin allgemein von Operationen.

auch Klassen sowie den mengenwertigen Typ `Set<·>` als Parametertypen bei der Spezifikation von Komponenten zu. In Anlehnung an die Templates der objektorientierten Programmiersprache C++ spezifiziert das syntaktische Konstrukt `Set<T>` durch die Parametrisierung des `Set`-Datentyps mit einem Typ `T` eine Menge von Werten des Typs `T`. Die Richtung eines Parameters wird durch die Schlüsselworte `in` für Eingabeparameter, `out` für Ausgabeparameter und `inout` für Ein- und Ausgabeparameter gekennzeichnet.

Neben diesen syntaktischen Elementen zur Beschreibung struktureller Eigenschaften von Komponenten bietet CDL auch Beschreibungselemente für die Spezifikation des Verhaltens einer Komponente und ihrer Klassen. Die Beschreibung des Verhaltens basiert dabei auf der Idee, die Spezifikation des *aktiven Verhaltens*, d. h. die ausgelösten Operationsaufrufe, von der Spezifikation des *passiven Verhaltens*, d. h. der Bereitschaft zum Empfang solcher Aufrufe (*readiness*, vgl. [Old91]), zu trennen. Analog zur Kommunikation im π-Kalkül wird das Senden eines Operationsaufrufs durch den Bezeichner der Operation gefolgt von einem „!“ gekennzeichnet, die Bereitschaft zum Empfang eines solchen Aufrufs durch ein dem Bezeichner folgendes „?“. Eine Operation wird ausgeführt, wenn das Senden eines Operationsaufrufs durch eine Komponente mit der Empfangsbereitschaft einer entsprechenden Operation einer Zielkomponente zusammenfällt. In CDL spezifizieren wir das aktive Verhalten in Operationsrümpfen, während das passive Verhalten in speziellen Verhaltensbeschreibungen dokumentiert wird.

Zur Beschreibung des aktiven Verhaltens stellt CDL Konstrukte für den Aufruf von Operationen, die Erzeugung von Referenzen auf neue bzw. bereits existierende Geschäftsobjekte, (nichtdeterministische) Entscheidungen, Wiederholungsblöcke und die Rückgabe von Parameterwerten zur Verfügung. Die Beschreibung des aktiven Verhaltens einer Komponente in Operationsrümpfen hat primär illustrierenden Charakter. Sie wird im praktischen Einsatz oftmals als starke Unterspezifikation des tatsächlichen Verhaltens angegeben werden und eignet sich insbesondere dazu, verwendete Algorithmen zu dokumentieren und Abhängigkeiten von anderen Komponenten explizit darzustellen. Sieht man von der Erzeugung von Objektreferenzen ab, unterliegt der Beschreibung des aktiven Verhaltens in CDL keine formale Semantik. Die Beschreibung der Komponente `AccountComponent` macht in der Prozedur `openAccount()` Gebrauch vom `new`-Operator zur Erzeugung einer Referenz auf ein neues Geschäftsobjekt der Klasse `Account` (der alternative Operator zur Erzeugung von Referenzen auf existierende Geschäftsobjekte heißt `existing`; beide Operatoren werden später detaillierter beschrieben). Dabei wird dem `new`-Operator zum einen der voll qualifizierte Klassenname `AccountComponent.Account` und ein Bezeichner übergeben, über den das Geschäftsobjekt im Folgenden angesprochen werden kann. Die `return`-Anweisung bewirkt das Verlassen einer Operation. Je nachdem, ob die Operation `out`- oder `inout`-Parameter deklariert, können der `return`-Anweisung Argumente übergeben werden (ist der Wert eines Rückgabeparameters für die Aussagekraft der Komponentenbeschreibung irrelevant, kann der „Don't Care“-Wert `*` zurückgegeben werden). So gibt die Prozedur `openAccount()` der `AccountComponent` die Referenz auf das neu erzeugte `Account`-Objekt zurück, während z. B. die Methode `deposit()` der Klasse `Account` keinen Rückgabewert liefert. Weitere Konstrukte zur Beschreibung des aktiven Verhaltens werden im Folgenden anhand weiterer Komponenten erläutert.

Die Verhaltensbeschreibungen zur Spezifizierung des passiven Verhaltens einer Komponente bzw. einer Klasse basieren auf Protokollmaschinen, die wie alle Automaten-

modelle aus einer Menge von Zuständen sowie Transitionen zwischen diesen Zuständen bestehen. Zustände beschreiben dabei sinnvolle Abstraktionen von den tatsächlichen Zuständen, in denen sich eine Komponente oder ein Geschäftsobjekt befinden kann, während Transitionen die Operationsaufrufe darstellen, die eine Komponente oder Geschäftsobjekt von einem Zustand in einen Folgezustand überführen. Jede CDL-Verhaltensbeschreibung umfasst zumindest die zwei vorgegebenen Zustände `created` und `initialized`. Initial ist eine neu gestartete Komponenteninstanz bzw. ein neu erzeugtes Geschäftsobjekt im Zustand `created`. Dieser Zustand kann mit einem Initialisierungsskript (ähnlich einem Konstruktor in der objektorientierten Programmierung) verbunden sein, das aus einer Sequenz von Operationsaufrufen besteht, d. h. im Anfangszustand einer Verhaltensbeschreibung ist ausschließlich die Spezifikation aktiven Verhaltens zugelassen. Es gilt die Konvention, dass der Zustand `created` nur bei der Initialisierung des Objekts eingenommen wird und keine Operation die Protokollmaschine zu einem späteren Zeitpunkt wieder in diesen Zustand überführt. Nach erfolgter Initialisierung geht eine Komponente bzw. ein Geschäftsobjekt in den Folgezustand `initialized` über, der den effektiven Anfangszustand des eigentlichen Lebenszyklus repräsentiert. Beschreibungen des Verhaltens, das in diesem oder einem Folgezustand möglich ist, dürfen nur noch die Bereitschaft zum Empfang von Operationsaufrufen, d. h. das passive Verhalten der Komponente oder des Geschäftsobjekts beschreiben.

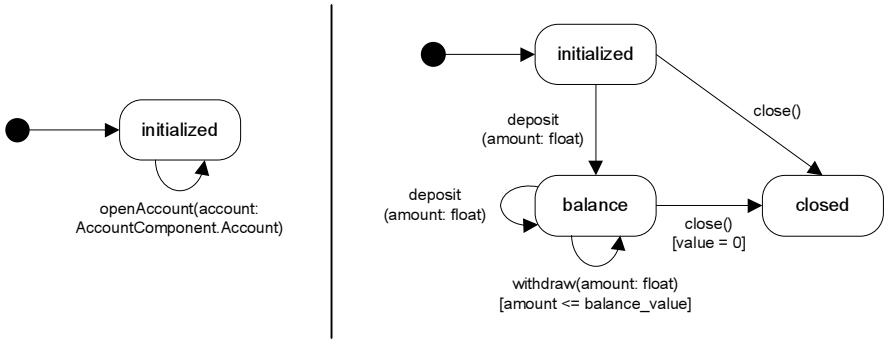


Abbildung 6.5: Verhalten der Komponente `AccountComponent` und der Klasse `Account`

Abbildung 6.5 stellt das in Listing 6.1 textuell spezifizierte passive Verhalten der Komponente `AccountComponent` und der Klasse `Account` als Protokollmaschine graphisch dar. Sowohl die Komponente `AccountComponent` als auch die Klasse `Account` erfordern keinerlei Initialisierung und gehen daher direkt vom Zustand `created` in den Zustand `initialized` über. `AccountComponent` ist im Zustand `initialized` bereit, Aufrufe der Prozedur `openAccount()` entgegenzunehmen. Bei Aufruf dieser Prozedur wird eine neue Instanz der Klasse `Account` erzeugt und über den `out`-Parameter `account` zurückgegeben. Anschließend kehrt `AccountComponent` wieder in den Zustand `initialized` zurück. Das neue `Account`-Geschäftsobjekt ist initial u. a. bereit, Einzahlungen entgegenzunehmen. Wird durch Aufruf der Methode `deposit()` eine Einzahlung über einen Betrag `amount` vorgenommen, geht das Geschäftsobjekt in den Folgezustand `balance(amount)` über. Da die Parametrisierung von Zuständen in den Protokollmaschinen der UML nicht

vorgesehen ist, weist der Zustand `balance` in der Repräsentation als (endliche) Protokollmaschine in Abbildung 6.5 keinen Parameter auf. Die Abweichung der CDL von den Protokollmaschinen der UML erscheint aus zwei Gründen sinnvoll. Zum einen lassen sich über die Parametrisierung von Zuständen Variablen definieren, die in den Beschreibungen der im jeweiligen Zustand aufrufbaren Operationen ähnlich wie globale Variablen benutzt werden können. So könnte sich die Beschreibung des aktiven Verhaltens der Methode `withdraw()` auf das Zustandsattribut `value` des Zustands `balance` beziehen. Zum anderen helfen Parameter, die fachliche Bedeutung der gewählten Zustände zu verdeutlichen sowie die durch die Komponente realisierte Geschäftslogik näher zu beschreiben. Es ist allerdings festzuhalten, dass CDL-Verhaltensbeschreibungen durch die Parametrisierung von Zuständen mit Werten aus möglicherweise unendlichen Domänen die Klasse der endlichen Automaten verlassen.

Alternativ zur Einzahlung eines Betrags kann das `Account`-Geschäftsobjekt im Zustand `initialized` durch Aufruf der `close()`-Methode geschlossen werden. Die Auswahl zwischen `deposit()` und `close()` ist *global nichtdeterministisch*, d. h. sie hängt einzig vom Interaktionsverhalten des Clients ab, und wird durch den `+`-Operator spezifiziert. Nach dem Aufruf der `close()`-Methode geht das Geschäftsobjekt in den Folgezustand `closed` über. Da das in diesem Zustand mögliche Verhalten durch den leeren Prozessterm `()` beschrieben wird, erlaubt die `Account`-Instanz keine weiteren Interaktionen.

Im Zustand `balance` ist das `Account`-Geschäftsobjekt bedingt bereit, durch Aufrufe der Methode `withdraw()` Abbuchungen vorzunehmen. Der Betrag einer Abbuchung ist dabei durch den aktuellen Kontostand begrenzt. Diese bedingte Bereitschaft zur Ausführung einer Operation wird in CDL durch das syntaktische Konstrukt `on <call> if <condition> do <statechange>` beschrieben, dessen Semantik auf den *Call Events* [OMG03] der UML-Zustandsmaschinen basiert. Im Falle des Aufrufs einer Operation `<call>` wird zunächst die Bedingung `<condition>` geprüft, die durch einfache boolesche Ausdrücke über Basisdatentypen mit den Operatoren `==`, `<`, `>`, `<=`, `>=` und `not` definiert wird. Ist diese Bedingung erfüllt, wird die Operation `<call>` ausgeführt und die Komponenteninstanz bzw. das Geschäftsobjekt wechselt in den durch `<statechange>` spezifizierten Zustand.

In Abschnitt 9.1.2 stellen wir die hier informell vorgenommene Abbildung der syntaktischen CDL-Konstrukte für die Beschreibung des passiven Verhaltens von Komponenten und Klassen (bzw. Kontrakten und Objekttypen) auf die Metaklassen der UML-Protokollmaschinen vor. In diesem Zusammenhang soll angemerkt werden, dass die Semantik der UML durch die OMG bislang nicht eindeutig spezifiziert wurde und ihre Definition Gegenstand verschiedener Forschungsarbeiten ist (vgl. z. B. [EFLR98, BCR00b]), die zum Teil in der *precise UML Group (pUML)* [PUM03] zusammengefasst werden.

Kontrakte Die CDL-Syntax für die Beschreibung von Kontrakten unterscheidet sich kaum von der für die Beschreibung geschlossener atomarer Komponenten. Lediglich die für die Beschreibung von Komponenten verwendeten Schlüsselwörter `component` und `class` müssen bei der Beschreibung von Kontrakten durch die Schlüsselwörter `contract` und `objecttype` ersetzt werden. Listing 6.2 zeigt die Beschreibung eines Kontrakts namens `AccountContract`, der Anforderungen an Komponenten für grundlegende Funktionalität zur Kontoverwaltung spezifiziert.

```
contract AccountContract {
    objecttype AccountType {
        deposit(in amount: float) {
            return;     /* Erhöhung des Kontostands nicht spezifiziert */
        };
        withdraw(in amount: float) {
            return;     /* Reduzierung des Kontostands nicht spezifiziert */
        };

        created()              =   initialized();
        initialized()          =   deposit?(amount).balance(amount);
        balance(value: float) =   deposit?(amount).balance(value+amount)
                                + on withdraw?(amount)
                                    if (amount <= value)
                                      do balance(value-amount);
    };

    openAccount(out account: AccountContract.AccountType) {
        new(AccountContract.AccountType account);
        return(account);
    };

    created()              = initialized();
    initialized()          = openAccount?(account).initialized();
};
```

Listing 6.2: Kontrakt `AccountContract`

Dieser Kontrakt unterscheidet sich von der Beschreibung der Komponente `Account-Component` neben den angesprochenen syntaktischen Merkmalen nur in zweierlei Hinsicht. Zum einen beinhaltet er nicht die Methode `close()`, d. h. eine Komponente, die diesen Kontrakt erfüllen soll, muss diese Methode nicht anbieten. Zum anderen verlangt die Prozedur `openAccount()` des Kontrakts die Erzeugung eines Geschäftsobjekts des Objekttyps `AccountContract.AccountType`, während die gleichnamige Prozedur der Komponente eine Instanz der Klasse `AccountComponent.Account` erzeugt.

Offene atomare Komponenten Offene atomare Komponenten unterscheiden sich von geschlossenen atomaren Komponenten dadurch, dass sie im Sinne eines Frameworks durch Komposition mit anderen Komponenten konfiguriert (parametrisiert) werden müssen, um einsetzbar zu sein. In CDL werden entsprechende Erweiterungspunkte durch das Schlüsselwort `slot` gekennzeichnet. Erweiterungspunkte spezifizieren durch Angabe eines Kontrakts indirekt die Komponenten, die mit der offenen Komponente kombiniert werden dürfen.

Listing 6.3 zeigt die Beschreibung einer offenen Komponente `BankComponent`, die einfache Dienste von Banken anbietet. Die `BankComponent` hat einen Erweiterungspunkt `accComp`, der mit Komponenten für die Verwaltung von Konten konfiguriert werden kann, die den Kontrakt `AccountContract` aus Listing 6.2 unterstützen. Neben diesem Erweiterungspunkt spezifiziert die offene Komponente `BankComponent` eine Klasse `Bank`,

```
component BankComponent {
    slot AccountContract accComp;

    class Bank {
        openAccount(out account: AccountContract.AccountType) {
            accComp.openAccount!(account);
            return(account);
        };
        transfer(in source: AccountContract.AccountType,
                 in dest: AccountContract.AccountType,
                 in amount: float) {
            choice {
                + [ source.withdraw!(amount) ] dest.deposit!(sum);
            };
            return;
        };

        created()     = initialized();
        initialized() = (  openAccount?(account)
                         + transfer?(source, dest, amount) )
                        .initialized();
    };

    createBank(out bank: BankComponent.Bank) {
        new (BankComponent.Bank bank);
        return(bank);
    };

    created()     = initialized();
    initialized() = createBank?(bank).initialized()
};
```

Listing 6.3: Offene atomare Komponente `BankComponent`

deren Instanzen über die Prozedur `createBank()` erzeugt werden können. Die Schnittstelle der Klasse `Bank` definiert Methoden `openAccount()` zum Eröffnen eines Kontos und `transfer()` für die Überweisung eines Betrags zwischen zwei Konten. In der Methode `transfer()` wird der `choice`-Operator eingesetzt, um auszudrücken, dass eine Gutschrift auf das Zielkonto nur dann erfolgt, wenn zuvor die Abbuchung vom Quellkonto erfolgreich war. In CDL gilt die Konvention, dass jeder Operationsaufruf implizit einen Wahrheitswert zurückliefert (*true*, wenn der Aufruf erfolgreich war, *false* sonst). Der `choice`-Operator stellt eine verallgemeinerte Form des üblichen `if-then-else`-Konstrukts dar, mit dem sich Entscheidungen zwischen einer beliebigen Menge von Alternativen spezifizieren lassen. Dabei sind auch nichtdeterministische Entscheidungen möglich.

Komplexe Komponenten Eine komplexe Komponente stellt eine Konfiguration einer offenen Komponente dar, bei der jeder Erweiterungspunkt mit einer Komponente parametrisiert ist, die den jeweils geforderten Kontrakt unterstützt. Betrachtet man offene Komponenten als *Templates*, so repräsentiert eine komplexe Komponente eine Instanz eines solchen Templates. In CDL werden komplexe Komponenten durch das Schlüsselwort

`complex` und die Angabe einer offenen Komponente sowie einer geordneten Liste von Komponenten für die Parametrisierung der Erweiterungspunkte spezifiziert. Eine komplexe Komponente exportiert die Schnittstellen der verwendeten offenen Komponente.

Listing 6.4 zeigt die Beschreibung einer komplexen Komponente `MyBankComponent`, die durch Komposition der offenen Komponente `BankComponent` mit der Komponente `AccountComponent` entsteht. Diese Komposition stellt eine gültige Konfiguration dar, da `AccountComponent` den vom Erweiterungspunkt `accComp` geforderten Kontrakt `AccountContract` unterstützt. `AccountComponent` gestattet jede von `AccountContract` geforderte Interaktionsfolge. Die Tatsache, dass ein Geschäftsobjekt der Klasse `Account` geschlossen werden kann, stellt einen Mehrwert dar, der aus der Perspektive der `BankComponent` irrelevant ist.

```
complex MyBankComponent = BankComponent(AccountComponent);
```

Listing 6.4: Komplexe Komponente `MyBankComponent`

Ist bereits zur Zeit der Entwicklung einer Komponente bzw. der Anfertigung einer Komponentenbeschreibung bekannt, dass die Komponente einen bestimmten Kontrakt unterstützt, so kann dies in CDL durch das Schlüsselwort `supports` ausgedrückt werden. Listing 6.5 zeigt die ersten Zeilen einer geänderten Fassung der Komponentenbeschreibung der `AccountComponent`, bei der die Unterstützung des Kontrakts `AccountContract` explizit gemacht wird. Eine solche explizite Spezifikation der Unterstützung eines Kon-

```
component AccountComponent supports AccountContract {
    class Account {
        ...
    };
};
```

Listing 6.5: Explizite Unterstützung eines Kontrakts

trakts ist allerdings nicht Voraussetzung dafür, dass ein Erweiterungspunkt mit einer Komponente parametrisiert werden kann.

Referenzen auf Geschäftsobjekte CDL stellt bei der Beschreibung von Komponenten den Lebenszyklus von Komponenteninstanzen und ihren Geschäftsobjekten in den Vordergrund. Diese zustandsorientierte Sichtweise spiegelt sich auch bei der Referenzierung von Geschäftsobjekten wider. Zu diesem Zweck betrachtet CDL zwei verschiedene Operatoren für die Erzeugung von Referenzen.

Der `new`-Operator beschreibt dabei den Fall, dass ein neu erzeugtes Geschäftsobjekt referenziert wird. Geschäftsobjekte, die mittels `new` referenziert werden, durchlaufen die ihrem `created`-Zustand zugeordnete Initialisierung und befinden sich danach im Zustand `initialized`. Das referenzierte Geschäftsobjekt gestattet damit die Interaktionen, die durch den `initialized`-Zustand spezifiziert werden.

Der Fall, dass ein bereits bestehendes, zu einem früheren Zeitpunkt angelegtes Geschäftsobjekt referenziert wird, wird duch den `existing`-Operator betrachtet. Dabei ist

zu berücksichtigen, dass vielfach keine Informationen über den Zustand des referenzierten Geschäftsobjekts vorliegen. Als Beispiel für diesen Fall können so genannte *Finder-Operationen* zum Suchen von Geschäftsobjekten in persistenten Datenbeständen dienen: Eine Operation zur Suche aller Aufträge eines bestimmten Kunden liefert eine Menge von Auftragsobjekten zurück, die im Allgemeinen keinen einheitlichen Auftragsstatus (z. B. „angenommen“, „geprüft“, „ausgeführt“ oder „storniert“) haben. CDL berücksichtigt dieses Problem durch die Annahme, dass ein mittels `existing` referenziertes Geschäftsobjekt mit Ausnahme des `created`-Zustands in jedem seiner Zustände sein kann. Es ermöglicht folglich jegliches Verhalten, das einem dieser Zustände zugeordnet ist. Die Menge der Zustände, in denen das Geschäftsobjekt potentiell sein kann, wird dabei durch folgende Interaktionen auf die Folgezustände derjenigen Zustände eingeschränkt, die diese Interaktionen gestatten.

Die Referenzierung existierender Geschäftsobjekte, deren Zustand bekannt ist (z. B. durch eine Finder-Operation zur Suche aller stornierten Aufträge), wird in CDL derzeit nicht berücksichtigt. Sie stellt eine durchaus interessante Ergänzung der Sprache dar, die jedoch weitreichende Auswirkungen auf die Erfüllung eines Kontrakts durch eine Komponente hat. Eine Komponente müsste nun nicht nur die im Kontrakt geforderten Interaktionsfolgen gestatten, sondern darüber hinaus auch eine bzgl. der referenzierten Zustände isomorphe Zustandseinteilung aufweisen.

6.2 Geschäftsprozessmodelle

In diesem Abschnitt behandeln wir die Anforderungsphase der Komponentensuche, in der Verwender ihre Anforderungen an geeignete Komponenten in Form von Geschäftsprozessmodellen formulieren. Dazu umreißen wir im Folgenden zunächst, welche der in Geschäftsprozessmodellen beschriebenen Aspekte betrieblichen Handelns durch die geschäftsprozessorientierte Komponentensuche berücksichtigt werden. Auf der Grundlage der resultierenden Anforderungen der geschäftsprozessorientierten Komponentensuche an eine Sprache zur Geschäftsprozessmodellierung begründen wir anschließend die Auswahl der linearen Prozessmodellierung für die Anfragespezifikation.

6.2.1 Geschäftsprozessmodelle als Grundlage der Komponentensuche

Geschäftsprozessmodelle beinhalten umfangreiche Informationen über die zentralen betrieblichen Abläufe innerhalb eines Unternehmens. Dabei variiert der Umfang der erfassten Informationen in Abhängigkeit von der für die Erstellung der Geschäftsprozessmodelle verwendeten Sprache (vgl. Abschnitt 2.2) sowie der mit der Modellierung von Geschäftsprozessen verfolgten Zielsetzung. So ist die Erfassung der Ausführungsdauern von Aktivitäten beispielsweise nur dann erforderlich, wenn die beschriebenen Geschäftsprozesse z. B. hinsichtlich Durchlaufzeiten analysiert und optimiert werden sollen. Unabhängig von dieser Variabilität existiert ein Kern zentraler betrieblicher Aspekte, die in beinahe jedem Geschäftsprozessmodell Berücksichtigung finden (vgl. auch Abschnitt 2.3). Dies sind neben der Ablauforganisation, also der zeitlich-logischen Abfolge betrieblicher Aktivitäten, die Zuordnung dieser Aktivitäten zu Einheiten der betrieblichen Aufbauorganisation sowie die für die Ausführung von Aktivitäten benötigten bzw. durch ihre

Ausführung erzeugten oder veränderten betrieblichen Ressourcen (Sach- und Dienstleistungen sowie Informationen) [Sch98b].

Grundsätzlich besteht für die geschäftsprozessorientierte Komponentensuche die Möglichkeit, alle diese Aspekte in den Suchprozess einfließen zu lassen:

- *Ablauforganisation*: Die durch ein Geschäftsprozessmodell spezifizierte zeitlich-logische Abfolge von Aktivitäten lässt sich mit den Lebenszyklen von Komponenten und Geschäftsobjekten vergleichen. Geeignete Komponenten unterstützen die Ausführung eines Geschäftsprozesses durch die bedarfsgerechte Bereitstellung von Operationen, die für die Ausführung der betrieblichen Aktivitäten relevante Dienste implementieren.

- *Aufbauorganisatorische Zuordnung*: Die Zuordnung von Aktivitäten zu Organisationseinheiten führt implizit zu einer Gruppierung von Aktivitäten. Informationen über diese Gruppenbildung kann im Rahmen der geschäftsprozessorientierten Komponentensuche genutzt werden, um Komponenten zu finden, die möglichst alle Aufgaben einer Organisationseinheit abdecken, anstatt diese Aufgaben auf mehrere Komponenten zu verteilen (vgl. hierzu auch [OLK99]).

- *Ressourcen*: Die von einer betrieblichen Aktivität genutzten, manipulierten oder erzeugten Ressourcen können neben der Bezeichnung der Aktivität gewissermaßen als Bestandteil ihrer „Signatur“ betrachtet werden. Sofern es sich bei diesen Ressourcen um Informationen handelt, also der Informationsfluss beschrieben wird, können diese Faktoren im Rahmen der geschäftsprozessorientierten Komponentensuche genutzt werden, um die Präzision bei der Zuordnung betrieblicher Aktivitäten zu Operationen durch den Vergleich der jeweiligen Signaturen zu erhöhen.

Da wir uns in dieser Arbeit vornehmlich mit der Berücksichtigung des Verhaltens bei der Komponentensuche befassen wollen, konzentrieren wir uns im Folgenden auf den Aspekt der Ablauforganisation. Die Betrachtung der aufbauorganisatorischen Zuordnung von Aktivitäten stellt eine durchaus interessante Ergänzung dar, die zur Verbesserung der Suchergebnisse beitragen kann. Eine solche Erweiterung würde jedoch den Rahmen der Arbeit sprengen, setzt sie doch eine intensive Auseinandersetzung mit der Auswahl und Gestaltung von Softwarearchitekturen sowie der betrieblichen Arbeitsorganisation voraus (vgl. hierzu auch [Gro03]). Die Betrachtung der Ressourcen von Aktivitäten eines Geschäftsprozessmodells im Sinne eines Signaturvergleichs schließlich stellt nach unserer Überzeugung eine zu starke Anforderung an die Übereinstimmung der Informationsflüsse von Geschäftsprozessmodellen und Komponenten dar, die den Lösungsraum in nicht vertretbarem Maße einschränkt.

6.2.2 Lineare Prozessmodelle zur Anfragespezifikation

Im vorangegangenen Abschnitt haben wir die in Geschäftsprozessmodellen beschriebenen ablauforganisatorischen Anforderungen in den Vordergrund der geschäftsprozessorientierten Komponentensuche gestellt und auf eine Betrachtung von aufbauorganisatorischer Zuordnung, Ressourcenzugriff oder gar Ausführungsdauern von Aktivitäten verzichtet. Aufgrund dieser Konzentration auf die Ablauforganisation kann grundsätzlich jede der in

Abschnitt 2.2 vorgestellten Sprachen für die Geschäftsprozessmodellierung zur Spezifikation von Suchanfragen eingesetzt werden (vgl. die Gegenüberstellung in Abschnitt 2.3). Die einzige Anforderung, die die geschäftsprozessorientierte Komponentensuche an eine Sprache zur Geschäftsprozessmodellierung stellt, ist die Bereitstellung syntaktischer Konstrukte für die Beschreibung von Sequenzen, alternativen und parallelen Verzweigungen sowie Schleifen, mit denen die eindeutige Modellierung grundlegender betrieblicher Abläufe als Netze von Aktivitäten ermöglicht wird. Für diese syntaktischen Konstrukte wird gefordert, dass ihre Semantik hinreichend präzise definiert ist.

Wir haben im Rahmen dieser Arbeit o. B. d. A lineare Prozessmodelle für die Spezifikation ablauforganisatorischer Anforderungen gewählt. Der Ansatz der linearen Prozessmodellierung definiert eine einfach strukturierte Diagrammsprache, die alle aus Sicht der geschäftsprozessorientierten Komponentensuche relevanten syntaktischen Konstrukte für die Modellierung der Ablauforganisation anbietet (Operatoren für sequentielle, alternative und parallele Komposition sowie Schleifen). Darüber hinaus weist der Ansatz mit seiner prozessalgebraischen Grundlage ein solides formales Fundament auf. Die alternativ betrachteten EPKs und Aktivitätsdiagramme haben zwar eine unerreichte Verbreitung gefunden, sie verfügen jedoch über keine hinreichende Formalisierung. Geschäftsprozessmodelle auf der Grundlage von EPKs und Aktivitätsdiagrammen weisen zudem oftmals Mängel hinsichtlich der Wohlgeformtheit auf, wie sie das Metamodell der linearen Prozessmodellierung erzwingt. Die Tatsache, dass die aufbauorganisatorische Zuordnung und der Zugriff auf Ressourcen von Aktivitäten in linearen Prozessmodellen nur implizit modelliert werden können, fällt aus Sicht der geschäftsprozessorientierten Komponentensuche aufgrund der Beschränkung auf ablauforganisatorische Aspekte nicht ins Gewicht.

Neben diesen formalen Gründen haben allerdings auch pragmatische Betrachtungen Einfluss auf die Wahl linearer Prozessmodelle für die Anfragespezifikation gehabt. So hat die Möglichkeit bestanden, auf den Quellcode der Werkzeuge zur linearen Prozessmodellierung zuzugreifen. Dadurch haben sich gegenüber anderen Ansätzen zur Geschäftsprozessmodellierung Vorteile hinsichtlich der Integration und Evaluation der in dieser Arbeit entwickelten Konzepte zur Herstellung semantischer Vergleichbarkeit zwischen Aktivitäten eines Geschäftsprozessmodells und Operationen von Komponenten ergeben (siehe Kapitel 8). Darüber hinaus existiert für lineare Prozessmodelle ein einfaches Austauschformat auf Basis der Auszeichnungssprache *XML (Extensible Markup Language)*, das die Verarbeitung linearer Prozessmodelle in anderen Werkzeugen erleichtert.

Sofern die Spezifika linearer Prozessmodelle für die Ausführungen in den folgenden Kapiteln nicht ins Gewicht fallen, verzichten wir auf deren explizite Nennung und sprechen weiterhin verallgemeinernd von Geschäftsprozessmodellen.

6.3 Zusammenfassung

In diesem Kapitel haben wir zunächst mit CDL eine Sprache für die Beschreibung der Unternehmensschicht serverseitiger Fachkomponenten eingeführt. CDL basiert auf einem Komponentenmodell, das von den Spezifika der Komponententechnologien COM, EJB und CCM abstrahiert. Beim Entwurf der CDL standen zum einen die integrierte Beschreibung der Struktur und des Verhaltens von Komponenten im Vordergrund, zum anderen bestand eine zentrale Anforderung darin, fachlich orientierte Komponentenbeschreibun-

gen im Sinne „komponentenorientierter Softwarereferenzmodelle“ zu ermöglichen, die auch für Fachexperten verständlich sind.

Dem Anspruch, mit CDL eine Art „Softwarereferenzmodelle für Komponenten“ erstellen zu können, sind wir allerdings in diesem Abschnitt noch nicht in ausreichendem Maße gerecht geworden. Zwar gestattet CDL in der hier vorgestellten Form die Beschreibung der Lebenszyklen von Komponenteninstanzen und Geschäftsobjekten in Form von Protokollmaschinen. Allerdings definieren solche Protokollmaschinen nur eine *formale Semantik*, d. h. ein abstraktes (Maschinen-)Modell ohne Bezug zu einem konkreten Anwendungsgebiet. Diese formale Semantik ist noch um eine *fachliche Semantik* zu ergänzen, die dem abstrakten Modell durch Beschreibung der fachlichen Bezüge eine konkrete Interpretation zuordnet. In Kapitel 8 stellen wir den von uns verfolgten Ansatz zur Definition der fachlichen Semantik vor und zeigen, wie CDL-Beschreibungen um entsprechende semantische Annotationen ergänzt werden.

Im zweiten Abschnitt dieses Kapitels haben wir dargelegt, welche Aspekte des in Geschäftsprozessmodellen beschriebenen betrieblichen Handelns grundsätzlich bei der geschäftsprozessorientierten Komponentensuche Berücksichtigung finden könnten, und uns im Anschluss daran auf die Betrachtung ablauforganisatorischer Anforderungen eingeschränkt. Wir haben den Ansatz der linearen Prozessmodellierung für die Spezifikation von Geschäftsprozessmodellen im Rahmen der geschäftsprozessorientierten Komponentensuche gewählt, da dieser neben einer einfach strukturierten Modellierungssprache auf solider formaler Grundlage Vorteile im Hinblick auf die Integration und Evaluation unserer Konzepte geboten hat.

Kapitel 7

Geschäftsprozessorientierte Komponentensuche: Protokollvergleich

Die Untersuchung der Eignung einer Komponente für die Unterstützung eines Geschäftsprozesses im Rahmen der geschäftsprozessorientierten Komponentensuche basiert auf dem Vergleich abstrakter Beschreibungen auf der Protokoll- und Semantikebene. Dieses Kapitel widmet sich der Vorstellung von Konzepten zum Vergleich von Komponentenbeschreibungen und Geschäftsprozessmodellen auf der Protokollebene. Die Einordnung des Protokollvergleichs als Aktivität der Suchphase im Makroprozess der geschäftsprozessorientierten Komponentensuche ist in Abbildung 7.1 dargestellt.

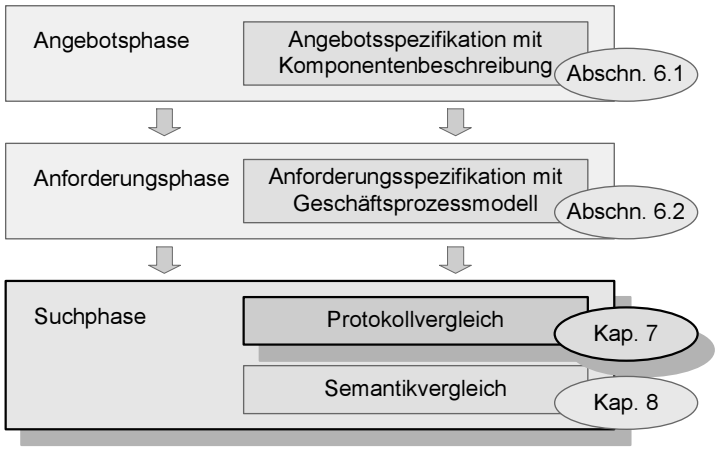


Abbildung 7.1: Einordnung des Kapitels in den Makroprozess

In Abschnitt 7.1 stellen wir ausgehend von einer übersichtsartigen Einführung in das Behavioural Subtyping unsere Betrachtung des Protokollvergleichs als Behavioural-Subtyping-Problem vor. Auf der Grundlage der in Abschnitt 7.2 aufgestellten Anforderungen an den Protokollvergleich untersuchen und bewerten wir in Abschnitt 7.3 verschiedene be-

kannte Behavioural-Subtyping-Relationen. Abschnitt 7.4 definiert eine formale Semantik für die dynamischen Aspekte von linearen Prozessmodellen einerseits und CDL-Komponentenbeschreibungen andererseits. Motiviert durch die Mängel existierender Subtyping-Relationen schlagen wir in Abschnitt 7.5 auf der Grundlage dieser formalen Semantik eine Adaption einer Subtyping-Relation für den Protokollvergleich sowie einen einfachen Algorithmus zur Berechnung dieser Relation vor. Abschnitt 7.6 beschließt das Kapitel mit einer Zusammenfassung.

7.1 Komponentensuche auf Basis des Behavioural Subtyping

In diesem Abschnitt geben wir zunächst eine kurze Einführung in das Konzept des Behavioural Subtyping, die auf der Habilitationsschrift von Wehrheim [Weh02] basiert. Im Anschluss gehen wir darauf ein, wie wir das Behavioural Subtyping für den Protokollvergleich im Rahmen der geschäftsprozessorientierten Komponentensuche konkret einsetzen wollen.

7.1.1 Einführung in das Behavioural Subtyping

Das Konzept des *Behavioural Subtyping* ist motiviert durch das Bestreben, inkrementelle Softwareentwicklungsprozesse [Bro97] zu ermöglichen, bei denen ausgehend von einem initialen (Analyse-)Modell auf jeder Entwicklungsstufe ein Systemmodell erstellt wird, das einerseits entweder das Modell der vorangegangenen Entwicklungsstufe konkretisiert oder durch neue Eigenschaften erweitert und andererseits alle relevanten funktionalen Eigenschaften des abstrakteren Modells erhält.

Im objektorientierten Paradigma wird die inkrementelle Softwareentwicklung durch das Konzept der *Vererbung* unterstützt. Vererbung ermöglicht die Bildung von Unterklassen durch die Wiederverwendung von Code einer Oberklasse sowie die Änderung und Erweiterung der Funktionalität der Oberklasse durch die Unterklasse. Dabei verhindert das Konzept der Vererbung jedoch im Allgemeinen nicht, dass Änderungen in der Unterklasse vorgenommen werden, durch die funktionale Eigenschaften der Oberklasse verletzt werden. Eine Unterklasse kann somit ein Verhalten zeigen, das beträchtlich von dem der Oberklasse abweicht.

Die systematische Vermeidung der Verletzung von Eigenschaften beim Übergang zwischen Softwareartefakten unterschiedlicher Entwicklungsstufen ist Gegenstand der formalen Technik der *Verfeinerung*. Grundlegend für die Verfeinerung von Softwareartefakten ist ein *Prinzip der Substituierbarkeit*, nach dem ein (aktives) Objekt bzw. dessen Spezifikation durch eine Verfeinerung ersetzt werden können muss, ohne dass *irgendjemand* diese Ersetzung feststellen kann. Von *aktiven Objekten* spricht man dann, wenn Objekte als autonome Einheiten betrachtet werden, die einen eigenen Kontrollfluss definieren und mit anderen Objekten durch den Austausch von Nachrichten kommunizieren. Da die Bereitschaft zur Kommunikation bei aktiven Objekten durch spezielle Interaktionsprotokolle gesteuert wird, sind im Allgemeinen nicht alle Dienste eines aktiven Objekts zu jedem Zeitpunkt verfügbar (*non-uniform service availability*, siehe [Nie95]). Aus dem Prinzip der Substituierbarkeit für die Verfeinerung folgt, dass das Konzept der Verfeinerung die

Erweiterung der Funktionalität (z. B. durch Einführung neuer Dienste) verbietet. Aus diesem Grund eignet sich die Verfeinerung nicht als Grundlage eines inkrementellen Softwareentwicklungsprozesses.

Der Begriff des *Subtyping* in objektorientierten Formalismen kann als Kombination der Konzepte der Vererbung und der Verfeinerung betrachtet werden:

Subtyping „=“ Vererbung „+“ Verfeinerung

Das Subtyping gestattet somit einerseits die Erweiterung der Funktionalität durch eine Unterklasse und stellt andererseits sicher, dass relevante funktionale Eigenschaften der Oberklasse auch in der Unterklasse erhalten bleiben. Dabei ist anzumerken, dass Subtyp-Beziehungen zwischen Softwareartefakten grundsätzlich unabhängig von der tatsächlichen Existenz einer Vererbungsbeziehung sind. Die derzeit am weitesten verbreiteten objektorientierten Programmiersprachen *C++* und *Java* ziehen aus Komplexitätsgründen jedoch für die Bestimmung von Subtyp-Beziehungen lediglich die Vererbungshierarchie heran.

Ähnlich wie bei dem Konzept der Verfeinerung liegt auch dem Subtyping ein Prinzip der Substituierbarkeit zugrunde: Eine Instanz eines Subtyps soll in jedem *Kontext* genutzt werden können, in dem eine Instanz eines Supertyps erwartet wurde (übersetzt nach [WZ88], siehe auch Originalzitat in Abschnitt 5.3). Im Gegensatz zum Prinzip der Substituierbarkeit bei der Verfeinerung wird beim Subtyping also nicht gefordert, dass die Ersetzung eines Softwareartefakts überhaupt nicht feststellbar ist. Es wird lediglich postuliert, dass ein Client, der mit einer Instanz des Supertyps rechnet, dessen Ersetzung durch eine Instanz des Subtyps nicht feststellen kann. Da Subtyp-Relationen, die diese Form der Substituierbarkeit sicherstellen wollen, das Verhalten von Objekten betrachten müssen, wird vom *Behavioural Subtyping* gesprochen.[1]

Beim Behavioural Subtyping lassen sich zwei Ausprägungen unterscheiden. Das *zustandsbasierte Behavioural Subtyping* (*state-based behavioural subtyping*) betrachtet das Verhalten von Typen in Gestalt der Semantik ihrer Methoden. In den grundlegenden Arbeiten zum zustandsbasierten Behavioural Subtyping von Wegner und Zdonik [WZ88] sowie America [Ame91] wird die Semantik einer Methode m dazu durch eine *Korrektheitsformel* der Form $\{P\}\, m\, \{Q\}$ mit Vorbedingung P und Nachbedingung Q beschrieben (vgl. Abschnitt 3.4.2). Zentraler Gedanke dieser Ansätze ist, dass ein Subtyp alle Methoden eines Supertyps semantisch erhalten bzw. verfeinern muss. Dazu wird im Allgemeinen verlangt, dass der Subtyp alle Methoden des Supertyps anbietet, wobei er die Vorbedingungen abschwächen und die Nachbedingungen verschärfen darf. Liskov und Wing haben diesen Ansatz in [LW94] um die Betrachtung der Auswirkungen zusätzlicher, durch den Subtyp eingeführter Methoden erweitert. Ein Beispiel für die Umsetzung des zustandsbasierten Behavioural Subtyping in der objektorientierten Programmierung stellt die Sprache *Eiffel* dar, die dem Gedanken des *Design by Contract* folgend Vor- und Nachbedingungen von Methoden sowie Zustandsinvarianten von Klassen bei der Bestimmung von Subtyp-Eigenschaften berücksichtigt.

Anders als bei der zustandsbasierten Variante steht beim *verhaltensorientierten Behavioural Subtyping* (*behaviour-oriented behavioural subtyping*) nicht die Semantik der Me-

[1] Wird wie hier explizit zwischen *Behavioural Subtyping* und *Subtyping* differenziert, so bezeichnet letztgenannter Begriff einen Ansatz, bei dem semantische Aspekte außer Acht gelassen und lediglich Signaturen von Methoden betrachtet werden.

thoden eines Typs, sondern deren Verfügbarkeit im Zentrum des Interesses. Eine grundlegende Annahme des verhaltensorientierten Behavioural Subtyping ist dabei, dass die betrachteten Typen *aktive* Objekte mit eingeschränkter Verfügbarkeit ihrer Methoden beschreiben. Der zentrale Gedanke der verhaltensorientierten Ansätze zum Behavioural Subtyping ist daher, dass ein Subtyp mindestens dieselbe Verfügbarkeit der Methoden wie der Supertyp bieten muss. Dazu wird das dynamische Verhalten von Typen auf der Grundlage von Interaktionsprotokollen betrachtet, die die zeitliche Ordnung möglicher Methodenaufrufe beschreiben. Eine Möglichkeit, derartige Subtyp-Beziehungen zu überprüfen, liegt im Einsatz so genannter *Testszenarien*. Ein Testszenario definiert gewissermaßen als „Versuchsaufbau“ Rahmenbedingungen für die Prüfung der Verfügbarkeit der Methoden von Supertyp und mutmaßlichem Subtyp. Dabei werden beide Typen hinsichtlich ihres Verhaltens verglichen, indem wie in [DNH84] ein Test-Client (evtl. ergänzt um weitere Clients, die wie in [Weh02] die Umgebung darstellen) gleichermaßen mit Objekten beider Typen interagiert. Ein Typ ist dann ein Subtyp eines anderen Typen, wenn *kein* Test-Client einen Unterschied zwischen beiden Typen hinsichtlich des Interaktionsverhaltens feststellen kann, weil jegliche Deadlock-Situation bei der gleichen Interaktion zwischen dem Test-Client, der evtl. vorhandenen Umgebung und dem Subtyp auch bei der Interaktion mit dem Supertyp auftreten kann.

7.1.2 Protokollvergleich mittels Behavioural Subtyping

Im Kontext der geschäftsprozessorientierten Komponentensuche erachten wir sowohl den Vergleich der Semantik von Diensten (Semantikebene) als auch den Vergleich der Verfügbarkeit dieser Dienste (Protokollebene) für unverzichtbar. Da jedoch der semantische Vergleich auf der Grundlage von Vor- und Nachbedingungen keinen operationellen Charakter hat und daher für unsere Zwecke ungeeignet ist, verfolgen wir den Ansatz des *zustandsbasierten* Behavioural Subtyping nicht weiter. Stattdessen vergleichen wir die fachliche Semantik von Diensten auf einer sprachlichen Ebene (siehe Kapitel 8). Für den Vergleich der durch Geschäftsprozessmodelle und CDL-Komponentenbeschreibungen spezifizierten Protokolle greifen wir auf grundlegende Ideen des *verhaltensbasierten* Behavioural Subtyping zurück. Wenn wir im Folgenden also verkürzend von Behavioural Subtyping sprechen, so meinen wir stets dessen verhaltensorientierte Ausprägung.

In der einführenden Vorstellung unseres Ansatzes zur geschäftsprozessorientierten Komponentensuche in Abschnitt 5.3 haben wir Geschäftsprozessmodelle als Spezifikationen verhaltensmäßiger Anforderungen eines Supertyps in einem Behavioural-Subtyping-Problem betrachtet. Mit unserem Wissen über den Gedanken der Testszenarien können wir diese Darstellung nun konkreter fassen. In den von uns betrachteten Behavioural-Subtyping-Problemen spielen Geschäftsprozessmodelle genau genommen nicht die Rolle eines Supertyps, sondern vielmehr die der Spezifikation eines Test-Clients, der im Rahmen eines Testszenarios eingesetzt wird. Die Rolle des Supertyps nimmt eine imaginäre „ideale“ Komponente ein, d. h. eine Komponente, die den durch das Geschäftsprozessmodell spezifizierten Test-Client ohne Deadlocks bedienen kann und folglich die Anforderungen des Geschäftsprozessmodells exakt erfüllt. Da eine solche ideale Komponente im Allgemeinen nicht existiert bzw. zumindest nicht bekannt ist (sonst wäre das Suchproblem trivial), sollen mit der geschäftsprozessorientierten Komponentensuche Komponenten bestimmt

werden, die sich aus Sicht des Test-Clients nicht bzw. nur in möglichst geringem Maße von der idealen Komponente unterscheiden und somit als Subtyp der idealen Komponente betrachtet werden können. Die Tatsache, dass andere Test-Clients Unterschiede zwischen der idealen und der gefundenen Komponente feststellen könnten, ist aus Sicht der geschäftsprozessorientierten Komponentensuche nicht relevant. Abbildung 7.2 fasst diese Konkretisierung der geschäftsprozessorientierten Komponentensuche als Behavioural-Subtyping-Problem graphisch zusammen.

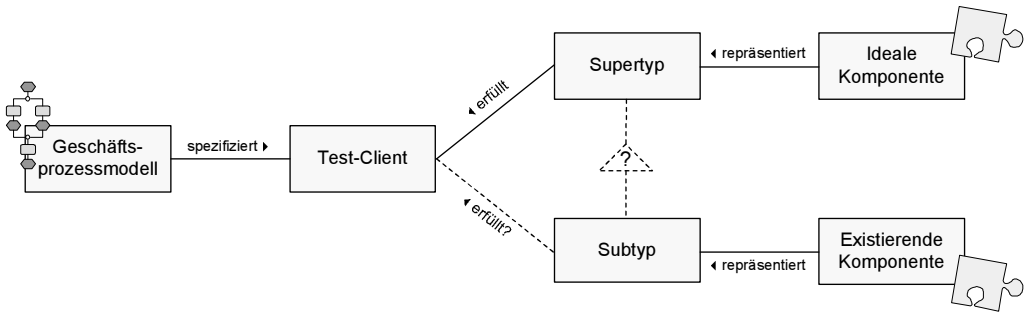


Abbildung 7.2: Geschäftsprozessmodell als Test-Client im Behavioural Subtyping

Um untersuchen zu können, ob eine Komponente einen Subtyp einer idealen Komponente darstellt, ist eine formale Beschreibung des *passiven* Verhaltens beider Komponenten erforderlich. Als Formalismus für die Beschreibung des passiven Verhaltens wählen wir in dieser Arbeit *beschriftete Transitionssysteme*, d.h. gerichtete Graphen mit beschrifteten Kanten. Das Verhalten der auf Subtyp-Eigenschaft untersuchten Komponente lässt sich als Netz beschrifteter Transitionssysteme darstellen, indem die direkte Schnittstelle der Komponente sowie jede ihrer indirekten Schnittstellen durch ein beschriftetes Transitionssystem repräsentiert wird. Dabei werden die in der CDL-Beschreibung spezifizierten Zustände auf Knoten und die Zustandsübergänge auf Kanten abgebildet, die mit dem Namen der aufgerufenen Operation beschriftet sind. Die Vernetzung der Transitionssysteme resultiert aus der Referenzierung von Geschäftsobjekten durch die mit den Transitionen verknüpften Operationen. Zur Illustration dieser Vernetzung von Transitionssystemen zeigt Abbildung 7.3 die Repräsentation des Verhaltens einer Komponente `OrderManagement`, die über eine Klasse `Order` Funktionalität zur Verwaltung von Aufträgen bereitstellt. Diese Komponente bietet insbesondere zwei Operationen `createOrder()` und `findOrder()` an, mit denen Aufträge in Gestalt von `Order`-Objekten erzeugt bzw. gesucht werden können. Während die Operation `createOrder()` den `new`-Operator verwendet und ein referenziertes Objekt sich daher im Anfangszustand `initialized` befindet, liegt der Operation `findOrder()` der `existing`-Operator zugrunde, d. h. ein referenziertes Objekt kann in jedem seiner möglichen Zustände (`initialized`, `approved`, `rejected` oder `end`) sein.

Da die ideale Komponente nicht bekannt ist und wir folglich nicht auf ihre CDL-Beschreibung zurückgreifen können, behelfen wir uns, indem wir uns eine abstrakte Komponente vorstellen, der wir das im Geschäftsprozessmodell geforderte Verhalten „unterstellen“ (nach Charakterisierung der idealen Komponente wissen wir, dass sie einen

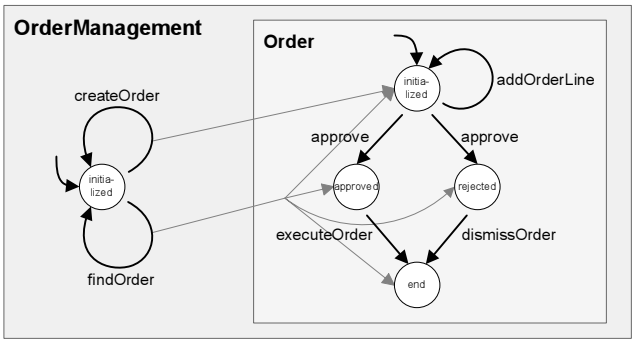


Abbildung 7.3: CDL-Verhaltensbeschreibung als System vernetzter Transitionssysteme

Test-Client bedienen kann, der eben dieses Verhalten fordert). Das Verhalten der idealen Komponente lässt sich daher darstellen, indem wir das geforderte Geschäftsprozessmodell als beschriftetes Transitionssystem interpretieren. Dazu bilden wir die Aktivitäten eines Geschäftsprozessmodells auf beschriftete Kanten ab und führen Knoten ein, um die Ablaufstruktur des Prozessmodells nachzubilden. Aufgrund dieser Interpretation erwähnen wir im Folgenden die idealen Komponenten nicht mehr und sprechen nur noch von Geschäftsprozessmodellen. Abbildung 7.4 zeigt exemplarisch ein einfaches Geschäftsprozessmodell zur Auftragsbearbeitung in seiner Darstellung als beschriftetes Transitionssystem.

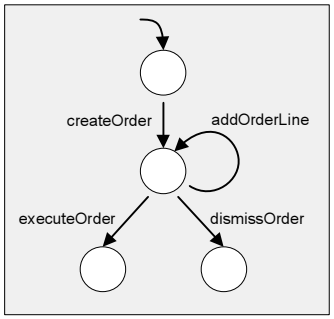


Abbildung 7.4: Geschäftsprozessmodell als beschriftetes Transitionssystem

Wie aus Abbildung 7.3 ersichtlich ist das Verhalten einer Komponente im Allgemeinen nicht durch ein einzelnes beschriftetes Transitionssystem gegeben, sondern vielmehr durch eine Menge solcher Systeme. Für die Bestimmung von Subtyp-Beziehungen zwischen Geschäftsprozessmodellen und Komponenten ist es daher nicht ausreichend, „einfach“ zwei Transitionssysteme miteinander zu vergleichen. Ein Kerngedanke unseres Ansatzes zur geschäftsprozessorientierten Komponentensuche besteht aus diesem Grund darin, dass die „Durchführung“ der Aktivitäten eines Geschäftsprozessmodells unter Nutzung der Operationen der betrachteten Komponente zur dynamischen Erzeugung von Geschäftsobjektreferenzen führen kann. Da diese Objektreferenzen die weitere Ausführung des Geschäftsprozessmodells durch die von ihnen angebotenen Operationen unterstützen

können, sind auch die ihnen zugrunde liegenden Transitionssysteme (bzw. deren dynamisch erzeugte Instanzen) bei der Bestimmung von Behavioural-Subtyping-Beziehungen zu betrachten.

Als erläuterndes Beispiel sei angenommen, dass bei der Untersuchung der Behavioural-Subtyping-Beziehung zwischen dem Geschäftsprozessmodell aus Abbildung 7.4 und der Komponente `OrderManagement` aus Abbildung 7.3 die Aktivität `createOrder` mit der gleichnamigen Operation identifiziert wird. Da die Ausführung der Operation `createOrder()` eine Referenz auf eine Instanz der Klasse `Order` bzw. des entsprechenden Transitionssystems erzeugt, kann bei der weiteren Untersuchung davon ausgegangen werden, dass zur Ausführung der Folgeaktivitäten `addOrderLine`, `executeOrder` oder `dismissOrder` neben den Operationen, die dem `initialized`-Zustand der Komponente `OrderManagement` zugeordnet sind (`createOrder()` und `findOrder()`), auch die Operationen verfügbar sind, die im `initialized`-Zustand des neu erzeugten `Order`-Geschäftsobjekts zur Ausführung bereit sind (`addOrderLine()` und `approve()`).

7.2 Anforderungen an den Protokollvergleich

In Abschnitt 7.1.2 haben wir gefordert, dass eine Komponente, die eine Behavioural-Subtyping-Beziehung zu einem gegebenen Geschäftsprozessmodell aufweist, einen durch das Geschäftsprozessmodell spezifizierten Test-Client ohne Deadlocks bedienen kann und sich folglich in dieser Hinsicht nicht von einer idealen Komponente unterscheidet. Diese Forderung bedeutet im Einzelnen, dass die Komponente

1. Operationen anbietet, die die Ausführung der im Geschäftsprozessmodell beschriebenen Aktivitäten unterstützen,
2. den Aufruf dieser Operationen in der Reihenfolge gestattet, wie sie das Geschäftsprozessmodell verlangt, und
3. alternative sowie parallele Ausführungspfade des Geschäftsprozessmodells respektiert.

Die geschäftsprozessorientierte Komponentensuche folgt dabei der Idee des so genannten *may testing* [DNH84]: Es wird lediglich gefordert, dass eine Komponente das im Geschäftsprozessmodell geforderte Verhalten zeigen *kann*. Die Forderung des *must testing*, nach der eine Komponente das geforderte Verhalten garantieren *muss*, ist nach unserer Überzeugung für die Komponentensuche zu streng.

Komponentenhersteller, die ihre Komponenten auf einem Komponentenmarkt erfolgreich veräußern wollen, sind bei der Entwicklung von Komponenten gezwungen, eine möglichst gute Balance zwischen passgenauer Anwendbarkeit in spezifischen Anwendungskontexten und Wiederverwendbarkeit durch ein weites Kundenspektrum zu erreichen [BR89]. Aus diesem Grund erscheint es unwahrscheinlich, auf einem Komponentenmarkt eine Komponente zu finden, die die spezifischen, in einem Geschäftsprozessmodell formulierten Anforderungen exakt erfüllt. Zum einen entspricht die von einer Komponente angebotene Menge von Operationen selten den in einem Geschäftsprozessmodell geforderten Aktivitäten. Zum anderen genügen die durch eine Komponente ermöglichten Interaktionssequenzen nur in Ausnahmefällen genau den Anforderungen eines Geschäftsprozessmodells.

In der Regel lassen sich daher allenfalls verhaltensmäßige *Ähnlichkeiten* zwischen einer Komponente und einem Geschäftsprozessmodell feststellen. Um die eingangs genannten Anforderungen an die geschäftsprozessorientierte Komponentensuche ohne inakzeptablen Qualitätsverlust bei den Suchergebnissen abzuschwächen, berücksichtigen wir drei Arten von Abweichungen zwischen den durch ein Geschäftsprozessmodell und eine Komponentenbeschreibung spezifizierten Verhaltensprotokollen (vgl. auch [Tes01]).

In der folgenden informellen Charakterisierung dieser Abweichungen gehen wir o. B. d. A vereinfachend davon aus, dass das Verhalten von Komponenten durch ein einfaches beschriftetes Transitionssystem anstelle eines Netzes solcher Systeme beschrieben wird. Des Weiteren nehmen wir an, dass Operationen einer Komponente zu Aktivitäten eines Geschäftsprozessmodells aufgrund gleicher Bezeichnungen zugeordnet werden.

Embedding Eine Komponente kann neben dem von einem Geschäftsprozessmodell geforderten Verhalten zusätzliche bzw. erweiterte Interaktionssequenzen anbieten. So kann eine Komponente beispielsweise die Ausführung einiger „Initialisierungsoperationen“ erfordern, bevor sie bereit ist, die Ausführung der Aktivitäten des Geschäftsprozessmodells durch geeignete Operationen zu unterstützen. Während dieser Ausführung des Geschäftsprozessmodells kann die Komponente darüber hinaus alternative, vom Geschäftsprozessmodell abweichende Fortsetzungen der Interaktion ermöglichen. Und schließlich kann die Komponente nach Ausführung der Aktivitäten des Geschäftsprozessmodells die Ausführung weiterer Operationen anbieten. Um Komponenten, die derartiges zusätzliches Verhalten anbieten, nicht von der Menge relevanter Komponenten auszuschließen, schwächen wir die Anforderungen an die Übereinstimmung der Interaktionsprotokolle von Geschäftsprozessmodell und Komponente durch den Begriff des *Embedding* ab:

Definition 7.1 (Embedding) *Eine Komponente ist ein gültiger Kandidat für die Unterstützung der in einem Geschäftsprozessmodell festgehaltenen Aufgaben, wenn das dem Geschäftsprozessmodell zugeordnete beschriftete Transitionssystem in das Transitionssystem der Komponente eingebettet ist.*

Abbildung 7.5 zeigt ein Beispiel für Embedding, das wiederum aus dem Funktionalbereich „Auftragsverwaltung“ stammt. Komponente $\mathcal{C}_1$ bietet nicht nur Operationen für alle im Geschäftsprozessmodell $\mathcal{R}_1$ geforderten Aktivitäten, sondern respektiert auch die durch das Geschäftsprozessmodell vorgegebene Reihenfolge ihrer Ausführung sowie die Verzweigungsstruktur. Daneben bietet $\mathcal{C}_1$ auch Operationen und Interaktionssequenzen an, die nicht für die Ausführung des Geschäftsprozessmodells benötigt werden (gekennzeichnet durch gestrichelte Transitionen). Aufgrund der durch den Begriff des Embedding definierten Abschwächung unserer Anforderungen an die Übereinstimmung von Protokollen stellt die Komponente $\mathcal{C}_1$ einen gültigen Kandidaten für die Unterstützung des Geschäftsprozessmodells $\mathcal{R}_1$ dar.

Component Interference Die in Abbildung 7.5 dargestellte zusätzliche Funktionalität der Komponente $\mathcal{C}_1$ ist insofern „gutmütig“, als dass sie die Ausführung der Aktivitäten des Geschäftsprozessmodells $\mathcal{R}_1$ nicht behindert. Nach initialer Ausführung der Operation `checkStock()` können die Aktivitäten von $\mathcal{R}_1$ durch entsprechende Operation von

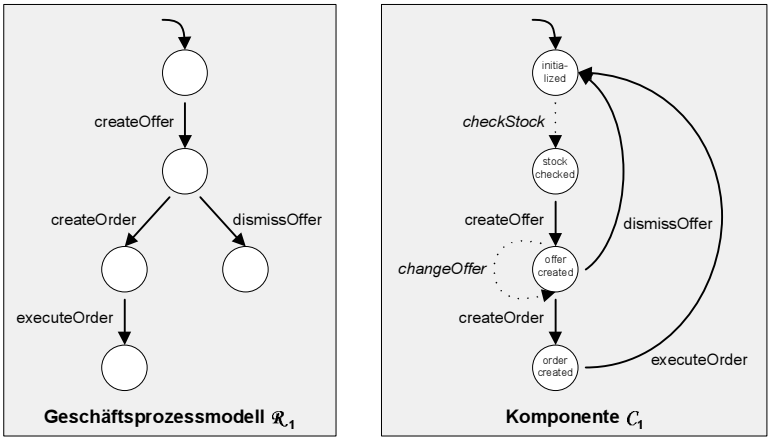


Abbildung 7.5: Beispiel für Embedding

$\mathcal{C}_1$ in direkter Folge ausgeführt werden. Der Fall, dass die Ausführung von Aktivitäten eines Geschäftsprozessmodells durch zusätzlich angebotene Operation einer Komponente unterbrochen wird, soll durch den Begriff der *Component Interference* berücksichtigt werden.

Definition 7.2 (Component Interference) *Eine Komponente ist ein gültiger Kandidat für die Unterstützung der in einem Geschäftsprozessmodell festgehaltenen Aufgaben, wenn das beschriftete Transitionssystem der Komponente nach Abstraktion von allen Operationen, die nicht durch das Geschäftsprozessmodell gefordert werden, die übrigen Anforderungen an die Komponentensuche erfüllt.*

Abbildung 7.6 stellt ein Beispiel für Component Interference dar. Die Komponente $\mathcal{C}_2$ fordert, dass mittels der Operation `validateOffer()` zunächst die Gültigkeit eines Angebots überprüft wird, bevor dieses durch die Operation `createOrder()` in einen Auftrag überführt wird. Da die Aktivität `validateOffer` nirgends in $\mathcal{R}_1$ gefordert wird, können wir sie als Unterspezifikation des Geschäftsprozessmodells anstatt als unerwünschte Funktionalität betrachten. Mit der durch den Begriff der Component Interference erreichten Abschwächung unserer Anforderungen stellt die Komponente $\mathcal{C}_2$ einen gültigen Kandidaten für die Unterstützung des Geschäftsprozessmodells $\mathcal{R}_1$ dar.

Process Interference Neben dem Fall, dass eine Komponente Funktionalität anbietet, die über die Anforderungen eines Geschäftsprozessmodells hinausgeht, kann auch der umgekehrte Fall vorliegen: Ein Geschäftsprozessmodell kann mehr Funktionalität fordern als von der betrachteten Komponente unterstützt wird. Analog zur Component Interference, also dem „Einstreuen“ zusätzlicher Operationsaufrufe in die Ausführung der Aktivitäten eines Geschäftsprozessmodells, kann ein Geschäftsprozessmodell folglich die Forderung nach der Ausführung zusätzlicher, nicht unterstützter Aktivitäten in das Interaktionsprotokoll einer Komponente „einstreuen“. Da die Implementierung der fachlichen Funktionalität in komponentenbasierten Softwaresystemen im Allgemeinen auf mehrere

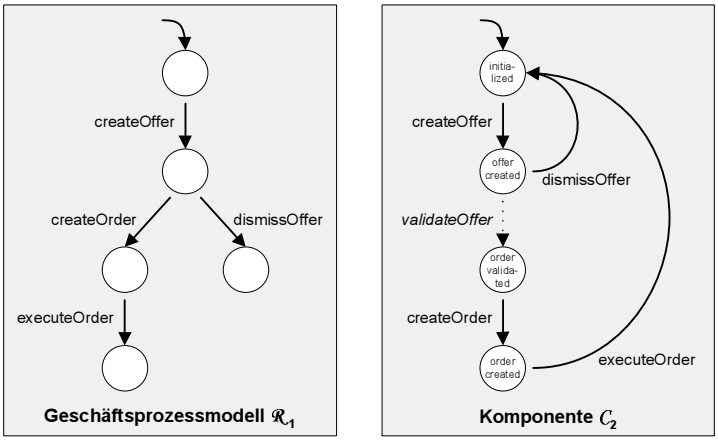


Abbildung 7.6: Beispiel für Component Interference

(miteinander kooperierende) Komponenten verteilt ist, nehmen wir angesichts dieses Falls an, dass eine andere Komponente des zu entwickelnden bzw. anzupassenden Softwaresystems geeignete Operationen zur Unterstützung dieser Aktivitäten bietet. Der Begriff der *Process Interference* schwächt unsere ursprünglichen Anforderungen entsprechend ab:

Definition 7.3 (Process Interference) *Eine Komponente ist ein gültiger Kandidat für die Unterstützung der in einem Geschäftsprozessmodell festgehaltenen Aufgaben, wenn sie die übrigen Anforderungen an die Komponentensuche bzgl. des Transitionssystems erfüllt, das aus dem Transitionssystem des Geschäftsprozessmodells durch Abstraktion von den Aktivitäten entsteht, die nicht von der Komponente unterstützt werden.*

Das Geschäftsprozessmodell $\mathcal{R}_2$ in Abbildung 7.7 spezifiziert, dass die Ausführung eines Auftrags zunächst zu planen ist (Aktivität `planOrderExecution`). Obwohl die Komponente $\mathcal{C}_3$ keine entsprechende Planungsfunktionalität zur Verfügung stellt, wird sie aufgrund der durch den Begriff der Process Interference erreichten Abschwächung der Anforderungen an die Übereinstimmung von Protokollen dennoch als sinnvoller Kandidat für die Unterstützung (von Teilen) des Geschäftsprozessmodells $\mathcal{R}_2$ betrachtet.

Zusammenfassend lassen sich die Anforderungen der geschäftsprozessorientierten Komponentensuche an den Protokollvergleich wie folgt formulieren:

> *Eine Komponente unterstützt die Ausführung der durch ein Geschäftsprozessmodell definierten Prozesse im Sinne der geschäftsprozessorientierten Komponentensuche, wenn sie die geforderten Reihenfolgen sowie die alternativen und parallelen Verzweigungen der Ausführung von Aktivitäten durch geeignete Operationen und ein entsprechendes Interaktionsprotokoll gestattet. Dabei sind bzgl. der Übereinstimmung der Protokolle Abweichungen gemäß Embedding, Component Interference und Process Interference zulässig.*

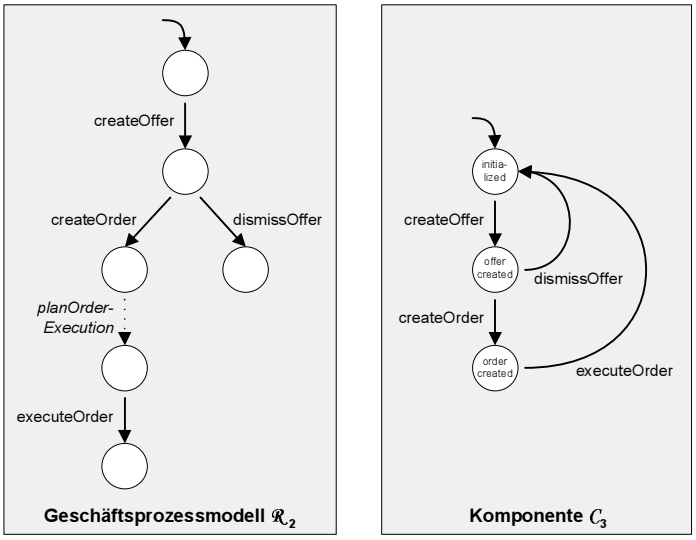


Abbildung 7.7: Beispiel für Process Interference

Diese Anforderungen charakterisieren gewissermaßen das Testszenario der geschäftsprozessorientierten Komponentensuche.

7.3 Analyse bekannter Behavioural-Subtyping-Relationen

In der Literatur auf dem Gebiet der Prozessalgebren sind eine Vielzahl von *Äquivalenzrelationen* und *Präordnungen*, also reflexive und transitive Relationen auf Transitionssystemen für den Einsatz im verhaltensorientierten Behavioural Subtyping vorgeschlagen worden. In diesem Abschnitt wollen wir einige der bekanntesten Präordnungen und Äquivalenzrelationen informell vorstellen und ihre Eignung für die geschäftsprozessorientierte Komponentensuche bewerten (vgl. auch [Kei01]). Eine umfassende formale Charakterisierung einer umfangreichen Sammlung von Präordnungen und Äquivalenzrelationen wird in [Gla90, Gla93] präsentiert.

Bei der Vorstellung und Bewertung der genannten Relationen betrachten wir wie im vorangegangenen Abschnitt nur Prozesse, die durch einfache beschriftete Transitionssysteme beschrieben sind, d. h. wir verzichten zugunsten einer klareren Präsentation zunächst auf die in Abschnitt 7.1.2 eingeführte Idee der dynamischen Erzeugung zusätzlicher Prozesse (und entsprechender Instanzen von Transitionssystemen). Wir gehen dabei davon aus, dass sich die vorgestellten Relationen entsprechend erweitern lassen. Diese Annahme lässt sich damit begründen, dass sich Mengen parallel ablaufender Prozesse mittels *Interleaving*-Semantik durch ein einzelnes Transitionssystem beschreiben lassen [Old91]. Für die beispielhafte Spezifikation von Prozessen nutzen wir neben „normalen“, *sichtbaren Aktionen*, die wir mit Buchstaben a, b, c, ... kennzeichnen, auch *unsichtbare* Aktionen. Da von Unterschieden zwischen verschiedenen unsichtbaren Aktionen hier zunächst abstra-

hiert werden kann, kennzeichnen wir unsichtbare Aktionen wie in Prozessalgebren üblich einheitlich mit dem Symbol τ. Soweit unsichtbare Aktionen nicht ausdrücklich genannt werden, beziehen sich unsere Ausführungen im Folgenden auf sichtbare Aktionen.

7.3.1 Trace Refinement und Trace Extension

Der Vergleich der ausführbaren Sequenzen von Aktionen, den so genannten *Traces*, stellt im Hinblick auf das Behavioural Subtyping den wohl nahe liegendsten Ansatz für den Vergleich von Prozessen dar. Er entspricht im Grunde dem *Wortproblem* [Sch92], bei dem zu entscheiden ist, ob ein gegebenes Wort über einer Menge von Aktionen in einer möglicherweise unendlichen Menge von Wörtern enthalten ist.

Saake et al. verlangen in ihrer Arbeit zu Vererbungsbeziehungen zwischen Objektlebenszyklen [SHJ+94], dass jeder mögliche Lebenszyklus eines Objekts einer Unterklasse auch einen gültigen Lebenszyklus eines Objekts der Oberklasse darstellt, wenn nur geerbte Ereignisse betrachtet werden. Die Traces eines Subtyps müssen also nach Projektion auf die Aktionen des Supertyps gültige Traces des Supertyps darstellen. Diese Verfeinerung der Trace-Menge des Supertyps durch den Subtyp bezeichnet man als *Trace Refinement* ($\sqsubseteq_{tr}$). Ebert und Engels unterscheiden in ihrer Arbeit zu Vererbungsbeziehungen zwischen Objektlebenszyklen [EE94] zwei Interpretationen von Verhaltensspezifikationen. Dabei bezeichnen sie mit dem Begriff des *Observable Behaviour* die Obergrenze des potentiell beobachtbaren Verhaltens eines Typs. Für eine Subtyp-Beziehung zwischen zwei Typen bzgl. des beobachtbaren Verhaltens fordern sie, dass der Subtyp die Trace-Menge des Supertyps gemäß Trace Refinement verfeinert. Für den Einsatz in der geschäftsprozessorientierten Komponentensuche erscheint der Ansatz des Trace Refinement nicht geeignet, da er die Erweiterung des Verhaltens des Supertyps durch den Subtyp nicht gestattet. Darüber hinaus werden keinerlei Anforderungen an den Erhalt des Verhaltens des Supertyps an den Subtyp gestellt. Insbesondere würde ein Prozess, der jegliche Ausführung von Aktionen verweigert, als Subtyp jeden Typs betrachtet werden.

Neben dem angesprochenen Observable Behaviour bezeichnen Ebert und Engels mit dem Begriff des *Invocable Behaviour* das von einem Typ im Sinne eines Kontrakts garantierte aufrufbare Verhalten. Für eine Subtyp-Beziehung zwischen zwei Typen bzgl. des aufrufbaren Verhaltens verlangen sie, dass der Subtyp alle Traces des Supertyps ermöglicht und folglich dessen Trace-Menge erweitert. Diese Erweiterung der Trace-Menge bezeichnet man als *Trace Extension* ($\sqsubseteq_{te}$). Den gleichen Ansatz macht auch Puntigam zur Grundlage seiner Arbeit zum Subtyping aktiver Objekte [Pun96]. Diese Variante der Berücksichtigung von Traces für die Bestimmung von Subtyp-Beziehungen erscheint allerdings ebenfalls nicht für die geschäftsprozessorientierte Komponentensuche geeignet. Abbildung 7.8 zeigt beschriftete Transitionssysteme zweier Prozesse $\mathcal{R}$ und $\mathcal{C}$, zwischen denen gemäß Trace Extension eine Subtyp-Beziehung besteht ($\mathcal{R} \sqsubseteq_{te} \mathcal{C}$)[2], d. h. $\mathcal{C}$ bietet alle Traces von $\mathcal{R}$.

Bemerkenswert ist dabei, dass $\mathcal{R}$ nach Ausführung der Aktion a die Ausführung der Folgeaktionen b und c ermöglicht, während $\mathcal{C}$ diese Alternative nicht anbietet. Da Trace

[2] In der Literatur zu Trace Extension wird allgemein die umgekehrte Notation verwendet. Um Irritationen beim Vergleich mit den üblichen Notationen anderer Präordnungen zu vermeiden, verwenden hier jedoch einheitlich das Format *Supertyp* $\sqsubseteq$ *Subtyp*.

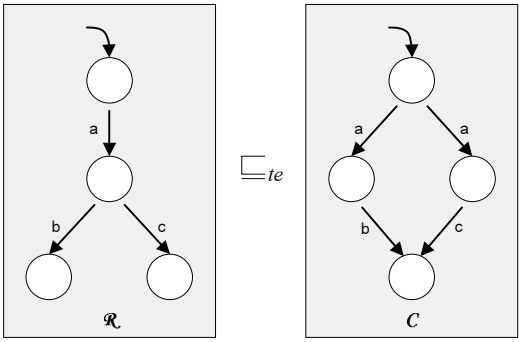


Abbildung 7.8: Beispiel für Trace Extension (nach Milner [Mil89])

Extension (ebenso wie Trace Refinement) nur Sequenzen von Aktionen betrachtet und nicht die Verzweigungsstruktur von Prozessen berücksichtigt, ist eine wichtige Anforderung an die Unterstützung eines Geschäftsprozessmodells durch eine Komponente verletzt (vgl. Abschnitt 7.2).

7.3.2 Reduction und Extension

Brinksma und Scollo haben in [BS86] mit *Reduction* und *Extension* zwei Subtyping-Relationen vorgeschlagen, die zentrale Mängel von Trace Refinement und Trace Extension beheben sollen. Grundlage beider Relationen ist die zusätzliche Betrachtung der nach einer durchgeführten Trace verweigerten Aktionen (*Refusals*).

Reduction ($\sqsubseteq_{red}$) ergänzt Trace Refinement um Lebendigkeitseigenschaften, indem gefordert wird, dass ein Subtyp nach einer beliebigen Trace weniger Aktionen verweigern darf als der Supertyp. Damit wird der in Abschnitt 7.3.1 betrachtete Fall ausgeschlossen, dass ein Subtyp die Ausführung jeglicher Aktionen verweigert. Allerdings verhindert Reduction wie schon Trace Refinement die Erweiterung des Verhaltens und ist damit für die geschäftsprozessorientierte Komponentensuche ebenfalls nicht geeignet.

Extension ($\sqsubseteq_{ext}$) gestattet wie Trace Extension die Erweiterung des Verhaltens durch einen Subtyp, betrachtet dabei allerdings auch die Verzweigungsstruktur von Prozessen. Ein Subtyp muss zum einen die Trace-Menge des Supertyps erweitern und darf zum anderen nach einer für den Supertyp gültigen Trace nicht mehr Aktionen verweigern als der Supertyp. Für die Prozesse $\mathcal{R}$ und $\mathcal{C}$ aus Abbildung 7.8 gilt folglich $\mathcal{R} \not\sqsubseteq_{ext} \mathcal{C}$. Extension stellt die Grundlage der Arbeit von Nierstrasz zum Subtyping aktiver Objekte [Nie95] dar. Abbildung 7.9 verdeutlicht, warum Extension nicht als Subtyping-Relation für die geschäftsprozessorientierte Komponentensuche geeignet erscheint.

Obwohl der Prozess $\mathcal{R}$ in den Prozess $\mathcal{C}$ eingebettet ist, stellt $\mathcal{R}$ gemäß Extension keinen Subtyp von $\mathcal{C}$ dar, d. h. $\mathcal{R} \not\sqsubseteq \mathcal{C}$. Der Prozess $\mathcal{C}$ garantiert aufgrund der nichtdeterministischen Entscheidung im Anfangszustand nicht, dass nach der Aktion a die Aktionen b und c ausführbar sind, wie es der Prozess $\mathcal{R}$ erwartet (die Aktion c ist ein Refusal des Prozesses $\mathcal{C}$ nach der Trace a). Extension realisiert also die Idee des *must testing*, d. h. ein Subtyp *muss* das Verhalten des Supertyps garantieren.

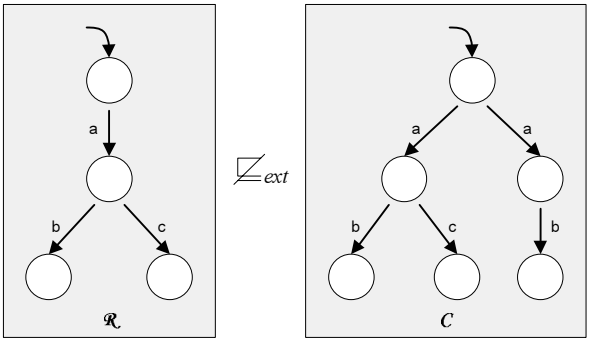


Abbildung 7.9: Extension gestattet nicht Embedding

7.3.3 Relationen auf Basis von Failures Refinement

Der grundlegende Gedanke der Verfeinerungsrelation *Failures Refinement* ($\sqsubseteq_{\mathcal{F}}$) [BHR84] besteht darin, dass ein Subtyp nicht mehr Interaktionen verweigern darf als ein Supertyp, um von einem Client nicht vom Supertyp unterschieden werden zu können. Die Menge der Fehlschläge (*Failures*) in der Kommunikation zwischen Client und Subtyp muss also eine Teilmenge der Kommunikationsfehlschläge sein, die zwischen Client und Supertyp beobachtet werden können. Failures Refinement wird als Korrektheitsrelation in der Prozessalgebra CSP [Hoa85] eingesetzt.

Da Failures Refinement nicht die Einführung neuer Aktionen durch den Subtyp gestattet, ist diese Relation in ihrer Grundform nicht als Grundlage des Protokollvergleichs bei der geschäftsprozessorientierten Komponentensuche geeignet. Fischer und Wehrheim schlagen in ihrer Arbeit zum Behavioural Subtyping in objektorientierten Systemen [FW00] vier Subtyping-Relationen auf Basis von Failures Refinement vor, die die Einführung neuer Aktionen durch den Subtyp ermöglichen. Dazu modifizieren sie Failures Refinement durch den Einsatz verschiedener Operatoren zur Behandlung neuer Aktionen.

Die Relation *Weak Subtyping* ($\sqsubseteq_{\mathcal{F}_r}$) stellt eine direkte Formalisierung der Subtyping-Idee dar: Solange ein Client eines Subtyps keine neuen Aktionen benutzt, soll dieser Typ das Verhalten des Supertyps aufweisen. Falls ein Client jedoch neue Aktionen nutzt, kann das nachfolgende Verhalten des Subtyps von dem des Supertyps beliebig abweichen. Die formale Definition von Weak Subtyping leitet sich aus Failures Refinement durch einen *Restriktionsoperator* ($\backslash_r$) ab, dessen Anwendung auf den Subtyp wie in Abbildung 7.10 dargestellt die Ausführung neuer Aktionen unterbindet. Da die resultierende Zerteilung von Prozessen in nicht erreichbare Teilprozesse auf direkte Weise der Anforderung der Component Interference widerspricht, eignet sich Weak Subtyping nicht für die geschäftsprozessorientierte Komponentensuche.

Safe Subtyping ($\sqsubseteq_{\mathcal{F}_s}$) greift die Idee von Liskov und Wing [LW94] auf, nach der die Wirkung neuer Aktionen durch die Wirkung von Sequenzen alter Aktionen „erklärt“ werden können muss. Damit zwei Typen gemäß Safe Subtyping in Beziehung stehen, wird verlangt, dass die Ausführung neuer Aktionen nur zu Zuständen führen kann, die auch durch die Ausführung von Sequenzen alter Aktionen erreicht werden können. Erreicht wird dies durch einen *Ersetzungsoperator*, der neue Aktionen des Subtyps durch

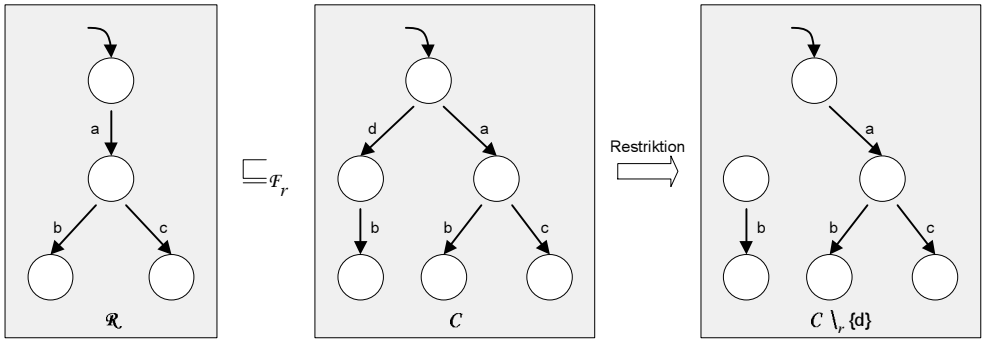


Abbildung 7.10: Anwendung des Restriktionsoperators

Folgen von Aktionen des Supertyps ersetzt. Dieser Ansatz erscheint aus dem Blickwinkel der hierarchischen Prozessmodellierung für die geschäftsprozessorientierte Komponentensuche durchaus interessant: Eine Komponente könnte eine Menge von Operationen anbieten, die miteinander kombiniert zur Unterstützung einer komplexeren Aktivität eines Geschäftsprozessmodells genutzt werden können. Bei dieser Anwendung werden also im Gegensatz zu dem Ansatz von Liskov und Wing „alte“ Aktivitäten durch „neue“ Operationen erklärt. Da allerdings die Zuordnung von Operationssequenzen zu Aktivitäten alles andere als offensichtlich ist, verfolgen wir diesen Ansatz nicht weiter.

Die Relationen *Optimistic Subtyping* und *Optimal Subtyping* basieren auf dem Gedanken, neue Aktionen des Subtyps zu verstecken, unterscheiden sich aber in dem dafür eingesetzten Operator. Beim *Optimistic Subtyping* ($\sqsubseteq_{\mathcal{F}_h}$) wird das klassische *CSP-Hiding* ($\setminus_h$) eingesetzt, bei dem Aktionen in unsichtbare τ-Aktionen umbenannt werden. Da Failures nur für *stabile* Zustände, d. h. Zustände ohne hinausführende τ-Aktionen berechnet werden, betrachtet Optimistic Subtyping folglich nur Zustände, in denen keine neuen Aktionen ausführbar sind. Dahinter verbirgt sich die *optimistische* Annahme, dass eine „freundliche“ Umgebung existiert, die mit dem Subtyp derart interagiert, dass ein Client, der den Supertyp erwartet, die durch das neue Verhalten begründeten Änderungen nicht wahrnimmt. Abbildung 7.11 zeigt ein Beispiel für die Anwendung des Hiding-Operators. Es gilt $\mathcal{R} \sqsubseteq_{\mathcal{F}_h} \mathcal{C}$, da das Verhalten im zweiten Zustand des Prozesses $\mathcal{C}\setminus_h\{d\}$ nicht in den Vergleich durch Failures Refinement eingeht. Optimistic Subtyping schwächt damit die Subtyping-Anforderung im Sinne von Component Interference ab, der symmetrische Fall der Process Interference wird jedoch nicht berücksichtigt.

Der Ansatz, nur das Verhalten in stabilen Zuständen zu betrachten, kann allerdings auch zu unerwünschten Effekten führen. Abbildung 7.12 zeigt zwei Prozesse $\mathcal{R}$ und $\mathcal{C}$, die trotz einer zusätzlichen Aktion d gemäß Component Interference in Beziehung stehen. Da Optimistic Subtyping jedoch das Verhalten des Prozesses $\mathcal{C}$ im zweiten (aufgrund von Hiding *instabilen*) Zustand nicht mehr berücksichtigt, ist die Ausführung der Aktion b nach der Aktion a anders als beim Prozess $\mathcal{R}$ nicht mehr möglich. Folglich besteht zwischen den Prozessen $\mathcal{R}$ und $\mathcal{C}$ gemäß Optimistic Subtyping keine Subtyp-Beziehung. Dieses Beispiel zeigt, dass Optimistic Subtyping von einer Umgebung ausgeht, die eine zusätzliche Aktion des Subtyps immer dann ausführt, wenn ihre Ausführung möglich

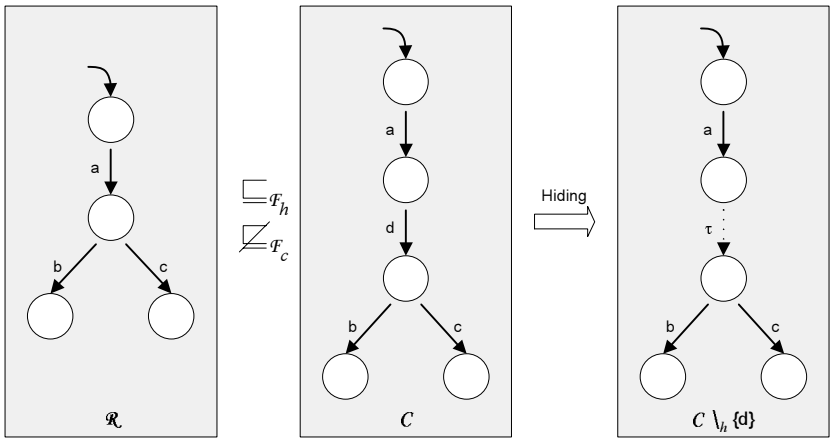


Abbildung 7.11: Anwendung des Hiding-Operators

ist, und nicht nur in den Situationen, in denen sie damit einem Client „behilflich“ wäre, der Folgeaktionen (in Abbildung 7.12 die Aktion c) dieser zusätzlichen Aktion ausführen möchte.

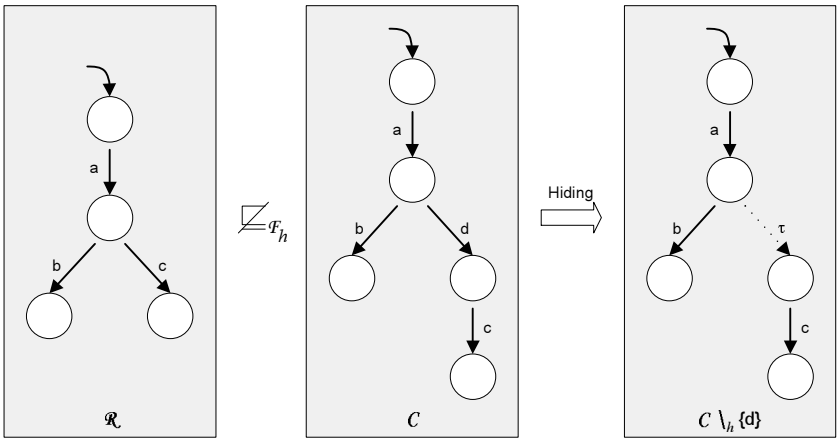


Abbildung 7.12: Optimistic Subtyping berücksichtigt nicht Component Interference

Optimal Subtyping ($\sqsubseteq_{\mathcal{F}_c}$) stellt die strengste der von Fischer und Wehrheim eingeführten Subtyping-Relationen dar. Ihr liegt die Vorstellung zugrunde, dass neue Aktionen das Verhalten des Supertyps im Subtyp nicht verändern dürfen. Im Unterschied zu Optimistic Subtyping wird nicht CSP-Hiding, sondern ein Operator namens *Concealment* eingesetzt. Dieser Operator benennt ebenfalls Aktionen in τ-Aktionen um, sorgt allerdings dafür, dass auch Failures in *instabilen* Zuständen betrachtet werden. Entsprechend gilt für die Prozesse $\mathcal{R}$ und C aus Abbildung 7.11 $\mathcal{R} \not\sqsubseteq_{\mathcal{F}_c} \mathcal{C}$. Dies erscheint aus Sicht der geschäftsprozessorientierten Komponentensuche zu streng, da der Begriff der Component Interference Veränderungen wie in der Abbildung dargestellt beschreibt.

7.3.4 Simulationsrelationen

Motiviert durch die mangelnde Berücksichtigung der Verzweigungsstruktur von Prozessen bei *Trace Extension* sind verschiedene Relationen entwickelt worden, die auf der Idee der *Simulation* ($\sqsubseteq_{sim}$) [Mil71] basieren (vgl. auch [Mil89]). Intuitiv besagt eine Simulationsrelation zwischen zwei Prozessen, dass jede Aktion des ersten Prozesses von dem zweiten Prozess derart „simuliert“ werden kann, dass zwischen den beiden verbleibenden Restprozessen ebenfalls eine Simulationsbeziehung besteht. Dabei gestattet Simulation, dass der simulierende Prozess zusätzliches Verhalten zeigt, solange nicht die Ausführung des geforderten Verhaltens behindert wird. Harel und Kupferman nutzen Simulation als Behavioural-Subtyping-Relation in ihrer Arbeit zur Vererbung von Lebenszyklen zwischen zustandsbasierten Objekten [HK00]. Eine Abschwächung der Simulationsanforderung wird durch *Weak Simulation* ($\sqsubseteq_{sim_\tau}$) erreicht, die neben sichtbaren Aktionen auch unsichtbare Aktionen berücksichtigt.

Milner und Park [Mil80, Par81] haben mit *(Strong) Bisimulation* ($\sim$) eine Äquivalenzrelation auf Prozessen definiert, die eine symmetrische Simulationsrelation zwischen Prozessen fordert. Durch die Symmetrieanforderung ist die Erweiterung des Verhaltens in Form neuer Aktionen anders als bei Simulation nicht möglich. Bisimulation wird als Äquivalenzrelation in der Prozessalgebra CCS [Mil89] eingesetzt. Sie bildet ferner die Grundlage der Arbeit von Canal et al. zur Kompatibilität und Verfeinerung von Verhalten im π-Kalkül [CPT99]. Analog zu Weak Simulation berücksichtigt *Weak Bisimulation* ($\sim_\tau$) [Mil89] neben sichtbaren Aktionen auch unsichtbare Aktionen. *Branching Bisimulation* ($\sim_b$) [GW96] setzt auf Weak Bisimulation auf und behebt deren Mängel bei der Berücksichtigung der Verzweigungsstruktur von Prozessen in Gegenwart unsichtbarer Aktionen. Van der Aalst et al. nutzen Branching Bisimulation als Grundlage ihrer Arbeiten zur Evolution von Workflows [AB02] und zur Kompatibilität in Softwarearchitekturen [AHT02].

In unserer Arbeit zur geschäftsprozessorientierten Komponentensuche sind wir nicht an Äquivalenzrelationen zwischen Prozessen interessiert, da diese zu starke Anforderungen an die Übereinstimmung zweier Prozesse stellen. Aus diesem Grund erscheinen von den hier vorgestellten Relationen zunächst einmal nur die Relationen Simulation und Weak Simulation für eine weitere Betrachtung relevant. Da Simulation (anders als ihre abgeschwächte Variante) jedoch nicht die Möglichkeit bietet, Aktionen zu ignorieren, die die Übereinstimmung von Prozessen im Sinne von Component Interference und Process Interference stören, stellt Simulation keine Option für den Protokollvergleich dar. Eine Alternative aus dem Bereich der Simulationsrelationen könnte eine einseitige, nicht symmetrische Variante der Branching Bisimulation darstellen, die wir hier als *Branching Simulation* ($\sqsubseteq_{sim_b}$) bezeichnen wollen. Branching Simulation ist wie ihre symmetrische Variante strenger als Weak Simulation. Abbildung 7.13 zeigt exemplarisch zwei Prozesse $\mathcal{R}$ und $\mathcal{C}$, die zwar gemäß Weak Simulation ($\mathcal{R} \sqsubseteq_{sim_\tau} \mathcal{C}$), nicht aber gemäß Branching Simulation ($\mathcal{R} \not\sqsubseteq_{sim_b} \mathcal{C}$) in Beziehung stehen.

Interessant aus Sicht der geschäftsprozessorientierten Komponentensuche ist hier, dass der Prozess $\mathcal{R}$ in den Prozess $\mathcal{C}$ eingebettet ist. Branching Simulation lässt also die Abschwächung der Übereinstimmung von Prozessen gemäß Embedding nicht zu. Allerdings berücksichtigt auch Weak Simulation die Einbettung von Prozessen nur dann wie ge-

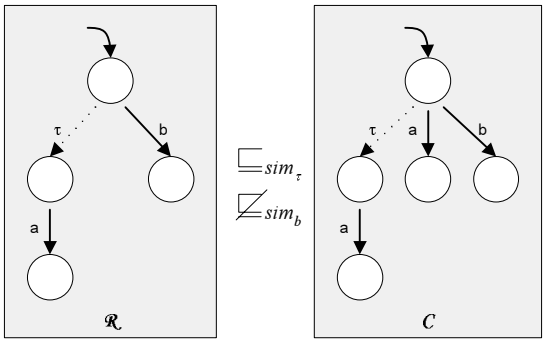


Abbildung 7.13: Gegenüberstellung von Weak Simulation und Branching Simulation

wünscht, wenn der eingebettete Prozess (im Beispiel $\mathcal{R}$) bereits zu Beginn des umfassenden Prozesses (im Beispiel $\mathcal{C}$) ausgeführt werden kann.

7.3.5 Bewertung

Tabelle 7.1 fasst die Bewertung der Behavioural-Subtyping-Relationen hinsichtlich der Berücksichtigung der Verzweigungsstruktur von Prozessen sowie der Unterstützung von Embedding, Component Interference und Process Interference zusammen. Dabei wird deutlich, dass keine der vorgestellten Relationen die Anforderungen der geschäftsprozessorientierten Komponentensuche vollständig erfüllt. Einige der untersuchten Relationen weisen jedoch Eigenschaften auf, die aus Sicht der geschäftsprozessorientierten Komponentensuche durchaus interessant erscheinen. So ähnelt die Annahme einer „freundlichen" Umgebung beim Optimistic Subtyping dem Gedanken der Component Interference, nach der zusätzlich angebotene Funktionalität einer Komponente u. U. doch akzeptabel ist. Das Verstecken neuer Aktionen des Subtyps stellt hier eine geeignete Realisierung dieser freundlichen Umgebung in der Subtyping-Relation dar. Allerdings gestattet Optimistic Subtyping nicht, durch das Verstecken von Aktionen des Supertyps eine ähnliche Wirkung im Sinne der Process Interference zu erzielen.

Der Ansatz des Optimal Subtyping, das Verhalten in instabilen Zuständen nicht gänzlich auszublenden, ist auch im Kontext der geschäftsprozessorientierten Komponentensuche sinnvoll. Die geschäftsprozessorientierte Komponentensuche fordert jedoch anstelle der bei Relationen auf Basis von Failures Refinement inhärenten pessimistischen Betrachtung der möglichen Fehlschläge im Sinne eines *must testing* eine positive Betrachtung des möglichen, aber nicht garantierten Verhaltens im Sinne eines *may testing*.

Weak Simulation unterstützt durch die Realisierung eines solchen *may testing* auch die Berücksichtigung von Embedding, wobei allerdings einschränkend verlangt wird, dass der Subtyp das Verhalten des Supertyps bereits von seinem Startzustand aus simulieren kann. Da Weak Simulation außerdem unsichtbare Aktionen auf Seiten des Super- und des Subtyps berücksichtigt, ist diese Relation geeignet, sowohl mit Component Interference als auch mit Process Interference umzugehen.

Relation	Struktur	Embedding	Component Interference	Process Interference
Trace Refinement	–	○ (Startzustand)	+ (Projektion)	–
Trace Extension	–	○ (Startzustand)	–	–
Reduction	+ (Refusals)	– (vgl. Abb. 7.9)	+ (Projektion)	–
Extension	+ (Refusals)	– (vgl. Abb. 7.9)	–	–
Weak Subtyping	+ (Failures)	– (vgl. Abb. 7.9)	–	–
Safe Subtyping	+ (Failures)	– (vgl. Abb. 7.9)	–	–
Optimistic Subtyping	+ (Failures)	– (vgl. Abb. 7.9)	○ (Hiding)	–
Optimal Subtyping	+ (Failures)	– (vgl. Abb. 7.9)	– (Concealment)	–
Simulation	+	○ (Startzustand)	–	–
Weak Simulation	+	○ (Startzustand)	+ (in Verbindung mit Hiding o. ä.)	+ (in Verbindung mit Hiding o. ä.)
Bisimulation	+	– (Äquivalenz-relation)	–	–
Weak Bisimulation	+	– (Äquivalenz-relation)	+ (in Verbindung mit Hiding o. ä.)	+ (in Verbindung mit Hiding o. ä.)
Branching Bisimulation	+	– (Äquivalenz-relation)	–	–
Branching Simulation	+	– (vgl. Abb. 7.13)	+ (in Verbindung mit Hiding o. ä.)	+ (in Verbindung mit Hiding o. ä.)

Tabelle 7.1: Bewertung der betrachteten Behavioural-Subtyping-Relationen

7.4 Formale Semantik von Geschäftsprozessmodellen und Komponentenbeschreibungen

Die Verwendung eines formalen Ansatzes zur Komponentensuche wie dem des verhaltensorientierten Behavioural Subtyping erfordert formale Beschreibungen des in Geschäftsprozessmodellen und Komponentenbeschreibungen spezifizierten Verhaltens. In diesem Abschnitt führen wir operationelle Definitionen der formalen Semantik von linearen Pro-

zessmodellen und CDL-Verhaltensbeschreibungen ein, auf deren Grundlage wir den Protokollvergleich durchführen.

7.4.1 Grundlagen

Als Formalismus für die Beschreibung der Semantik von linearen Prozessmodellen und CDL-Verhaltensbeschreibungen wählen wir wie bereits in Abschnitt 7.1.2 erwähnt *beschriftete Transitionssysteme*, d.h. gerichtete Graphen mit beschrifteten Kanten, die wir im Folgenden abkürzend als Transitionssysteme bezeichnen.

Sei Σ eine möglicherweise unendliche Menge von Aktionen, $\tau \notin \Sigma$ die unsichtbare Aktion und $\Sigma_\tau = \Sigma \cup \{\tau\}$. Sei ferner Θ eine möglicherweise unendliche Menge von (einfachen) Zuständen. Transitionssysteme können dann wie folgt definiert werden:

Definition 7.4 (Transitionssystem) *Ein (einfaches)* Transitionssystem *ist ein Tupel* $\mathcal{T} = (A, Q, \rightarrow, q_0)$*, wobei*

- $A \subseteq \Sigma$ *eine endliche Menge von Aktionen,*
- $Q \subseteq \Theta$ *eine endliche Menge einfacher Zustände,*
- $\rightarrow \subseteq Q \times (A \cup \{\tau\}) \times Q$ *eine Transitionsrelation und*
- $q_0 \in Q$ *der Startzustand ist.*

Die Menge von Aktionen A wird auch als Alphabet *des Transitionssystems ($\alpha(\mathcal{T})$) bezeichnet. Ein Element $(p, a, q) \in \rightarrow$ heißt* Transition *(beschriftet mit der Aktion a) und wird als $p \xrightarrow{a} q$ notiert. Wir bezeichnen die Menge aller Transitionssysteme mit der Abkürzung LTS für das englische „labelled transition system“.*

Für die graphische Illustration von Transitionssystemen und deren Transitionsverhalten verwenden wir eine hinsichtlich der Repräsentation von Zuständen an Petri-Netze angelehnte Darstellungsform. Dabei kennzeichnen wir aktive Zustände analog zur Markierung von Stellen in Petri-Netzen durch einen schwarzen Kreis innerhalb des den Zustand repräsentierenden Kreises. Abbildung 7.14 zeigt exemplarisch die Ausführung einer Transition $\xrightarrow{a}$.

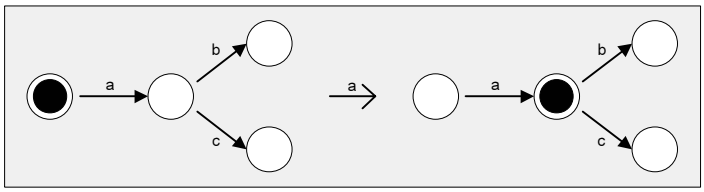


Abbildung 7.14: Einfache Transition mit Markierung des aktiven Zustands

Aufbauend auf der Transitionsrelation $\rightarrow$ definieren wir noch eine weitere Transitionsrelation $\Rightarrow$, die die Auswirkungen unsichtbarer Aktionen auf das sichtbare Transitionsverhalten eines Transitionssystems beschreibt.

Definition 7.5 (Abgeleitete Transitionsrelationen) *Sei* $\mathcal{T} = (A, Q, \rightarrow, q_0)$ *ein Transitionssystem,* $p, q \in Q$, $a \in \Sigma$ *und* $a_1, \ldots, a_n \in \Sigma_\tau$. *Dann gilt:*

- $p \xrightarrow{a_1 \cdots a_n} q \quad :\Leftrightarrow \quad \exists q_1, \ldots, q_{n+1} \in Q \quad \bullet \quad p = q_1$ *und* $q_i \xrightarrow{a_i} q_{i+1}$, $i \in \{1, \ldots, n\}$ *und* $q_{n+1} = q$
- $p \stackrel{a}{\Longrightarrow} q \quad :\Leftrightarrow \quad \exists p' \in Q, \sigma \in \{\tau\}^* \quad \bullet \quad p \xrightarrow{\sigma} p' \wedge p' \xrightarrow{a} q$

Dabei beschreibt $p \stackrel{a}{\Longrightarrow} q$ *die Möglichkeit, eine beliebige Anzahl von* τ*-Aktionen zu überlesen, bevor die gewünschte Aktion* a *ausgeführt wird.*

Abbildung 7.15 zeigt ein Beispiel für die Wirkung der Transitionsrelation $\Rightarrow$, in dem die gewünschte Aktion b nach der unbeobachteten Ausführung einer τ-Aktion durchgeführt wird.

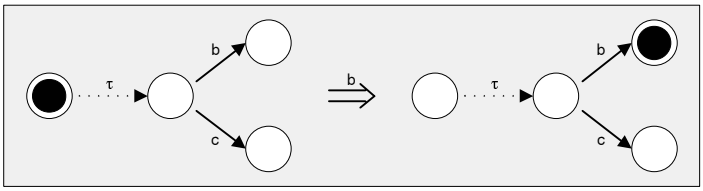


Abbildung 7.15: Einfache Transition mit unsichtbarer Aktion

7.4.2 Formale Semantik von linearen Prozessmodellen

Die formale Grundlage der Spezifikation linearer Prozessmodelle bilden Prozessterme, die auf einer Prozessalgebra mit Operatoren für sequentielle ($\mathcal{S}$), alternative ($\mathcal{A}$) und parallele Komposition ($\mathcal{P}$) sowie Iteration ($\mathcal{L}$) von Aktivitäten basieren (vgl. Abschnitt 2.2.3). Die Semantik dieser Operatoren, die in [Sch01, Sch03] lediglich informell beschrieben wird, soll im Folgenden durch eine operationelle Interpretation auf der Grundlage von Transitionssystemen formalisiert werden. Wir folgen dabei dem Ansatz von Plotkin, der mit der *Strukturierten Operationellen Semantik (SOS)* [Plo81] eine operationelle Interpretation von Prozesstermen der Prozessalgebra CSP definiert hat. Eine strukturierte operationelle Semantik ordnet einem Prozessterm P ein Transitionssystem zu, dessen Zustände die in der betrachteten Prozessalgebra zulässigen Prozessterme repräsentieren. Der Startzustand dieses Transitionssystems entspricht dabei dem Prozessterm P, für den eine operationelle Semantik anzugeben ist. Die Transitionsrelation des Transitionssystems ist induktiv über die Struktur von Prozesstermen definiert.

Sei $LProc$ die Menge der in der Prozessalgebra linearer Prozessmodelle zulässigen Prozessterme, die wir im Folgenden als lineare Prozessterme bezeichnen, P, P', Q, $Q' \in LProc$ lineare Prozessterme, $\epsilon \in LProc$ der leere Prozessterm, $a \in \Sigma_\tau$ eine Aktion und Ω ein spezielles Symbol, mit dem Roscoe [Ros97] in seiner Definition der operationellen Semantik von CSP die Terminierung eines Prozesses kennzeichnet. Roscoe betrachtet in

diesem Zusammenhang auch eine spezielle Terminierungsaktion $\surd$, nach deren Ausführung ein Prozessterm in den Endzustand Ω übergeht. Wir verzichten auf die Einführung dieser zusätzlichen Aktion und decken die Terminierung eines Prozesses durch die unsichtbare Aktion τ ab. Dann ist die Transitionsrelation der strukturierten operationellen Semantik linearer Prozessterme durch folgenden Kalkül gegeben:

(Elementarschritt)

$$a \xrightarrow{a} \epsilon \tag{7.1}$$

(Terminierung)

$$\epsilon \xrightarrow{\tau} \Omega \tag{7.2}$$

(Sequentielle Komposition)

$$\frac{P \xrightarrow{a} P'}{\mathcal{S}(P, Q_1, \ldots, Q_n) \xrightarrow{a} \mathcal{S}(P', Q_1, \ldots, Q_n)} \tag{7.3}$$

(Alternative Komposition)

$$\frac{P_i \xrightarrow{a} P_i' \quad (1 \leq i \leq n)}{\mathcal{A}(P_1, \ldots, P_i, \ldots, P_n) \xrightarrow{a} P_i'} \tag{7.4}$$

(Parallele Komposition)

$$\frac{P_i \xrightarrow{a} P_i' \quad (1 \leq i \leq n)}{\mathcal{P}(P_1, \ldots, P_i, \ldots, P_n) \xrightarrow{a} \mathcal{P}(P_1, \ldots, P_i', \ldots, P_n)} \tag{7.5}$$

(n-malige Iteration)

$n > 0$:

$$\frac{P \xrightarrow{a} P'}{\mathcal{L}(P) \xrightarrow{a} \mathcal{S}(P', \mathcal{L}(P))} \tag{7.6}$$

$n = 0$:

$$\mathcal{L}(P) \xrightarrow{\tau} \Omega \tag{7.7}$$

Neben diesen Regeln für die induktive Definition der Transitionsrelation gelten die folgenden Äquivalenzen, die das Terminierungsverhalten von sequentieller und paralleler Komposition beschreiben und der Vereinfachung von Termen dienen:

$$\mathcal{S}(\Omega, P_1, \ldots, P_n) = \mathcal{S}(P_1, \ldots, P_n) \quad (n \geq 1) \tag{7.8}$$

$$\mathcal{S}(P) = P \tag{7.9}$$

$$\mathcal{P}(\Omega, \ldots, \Omega) = \Omega \tag{7.10}$$

Gemäß Axiom 7.1 kann ein Prozessterm, der nur aus einer einzelnen Aktion besteht, durch deren Ausführung in den leeren Prozessterm übergehen. Axiom 7.2 beschreibt, dass der leere Prozessterm durch Ausführung einer τ-Aktion terminieren kann. Aus Regel 7.3 folgt, dass sich eine sequentielle Komposition $\mathcal{S}$ mehrerer Prozessterme dem Transitionsverhalten des ersten Prozessterms entsprechend weiterentwickeln kann. Nach der Terminierung der ersten sequentiellen Komponente kann sie gemäß Äquivalenz 7.8 aus der sequentiellen Komposition gelöscht werden. Zur Vereinfachung der resultierenden Terme kann die Äquivalenz 7.9 herangezogen werden. Regel 7.4 beschreibt das Transitionsverhalten der alternativen Komposition $\mathcal{A}$, bei der nicht gewählte Alternativen fallen gelassen werden. Der Sonderfall, bei dem eine der Alternativen leer ist und damit keinerlei Aktionen ausgeführt werden müssen, wird in linearen Prozessmodellen durch eine Alternative mit dem leeren Prozessterm ϵ modelliert, der gemäß Regel 7.2 direkt terminieren kann. Aus Regel 7.5 für die parallele Komposition $\mathcal{P}$ von Prozesstermen wird deutlich, dass es sich bei der angegebenen strukturierten operationellen Semantik um eine *Interleaving-Semantik* handelt. Eine Interleaving-Semantik modelliert Parallelität, indem alle möglichen Ausführungsfolgen der parallelen Zweige mittels sequentieller und alternativer Komposition „approximiert“ werden; sie gestattet jedoch nicht die Modellierung der tatsächlich parallelen (also gleichzeitigen oder zeitlich überlappenden) Ausführung von Aktionen. Die Äquivalenz 7.10 beschreibt, dass eine parallele Komposition nur dann terminieren kann, wenn alle ihrer parallelen Komponenten terminieren (*Und-Synchronisation*). Die Regeln 7.6 und 7.7 beschreiben die Semantik des Iterationsoperators $\mathcal{L}$, der die beliebige (und insbesondere nullmalige) Ausführung eines Prozesses spezifiziert. Mit diesen Regeln für die Transitionsrelation kann die Semantik linearer Prozessterme wie folgt definiert werden:

Definition 7.6 (Strukturierte operationelle Semantik linearer Prozessterme) *Die* strukturierte operationelle Semantik *von Prozesstermen der Prozessalgebra linearer Prozessmodelle ist eine Abbildung*

$$\mathcal{SOS}_{\mathcal{LP}}(\cdot) : LProc \longrightarrow LTS,$$

die jedem Prozessterm $P \in LProc$ ein Transitionssystem $\mathcal{T} \in LTS$ zuordnet mit

$$\mathcal{T} = (\alpha(P), LProc, \Rightarrow\!\downarrow_{\alpha(P)}, P),$$

wobei

$$\Rightarrow\!\downarrow_{\alpha(P)} = \{(P', a, Q') \mid P' \stackrel{a}{\Longrightarrow} Q' \wedge a \in \alpha(P)\}.$$

Dabei beschreiben $\alpha(P)$ die Menge aller Aktionen des linearen Prozessterms P (das Alphabet von P) und $\Rightarrow\!\downarrow_{\alpha(P)}$ die Einschränkung der Transitionsrelation auf Aktionen aus $\alpha(P)$ unter Berücksichtigung von τ-Aktionen.

Die Aktionen eines linearen Prozessterms entsprechen den Aktivitäten des durch den Prozessterm beschriebenen linearen Prozessmodells. Sofern nicht die tatsächliche Bezeichnung einer solchen Aktion mit ihrer fachlichen Semantik für das Verständnis wichtig erscheint, verwenden wir in dieser Arbeit verkürzend Aktionen a, b, c, ... als Platzhalter.

Abbildung 7.16 stellt exemplarisch das lineare Prozessmodell und die strukturierte operationelle Semantik des linearen Prozessterms

$$\mathcal{S}(a, \mathcal{A}(\mathcal{S}(b, c), \mathcal{S}(\mathcal{L}(\mathcal{S}(c, d)), e)), f)$$

gegenüber. In der Abbildung sind die Zustände des resultierenden Transitionssystems mit den durch sie repräsentierten Prozesstermen beschriftet.

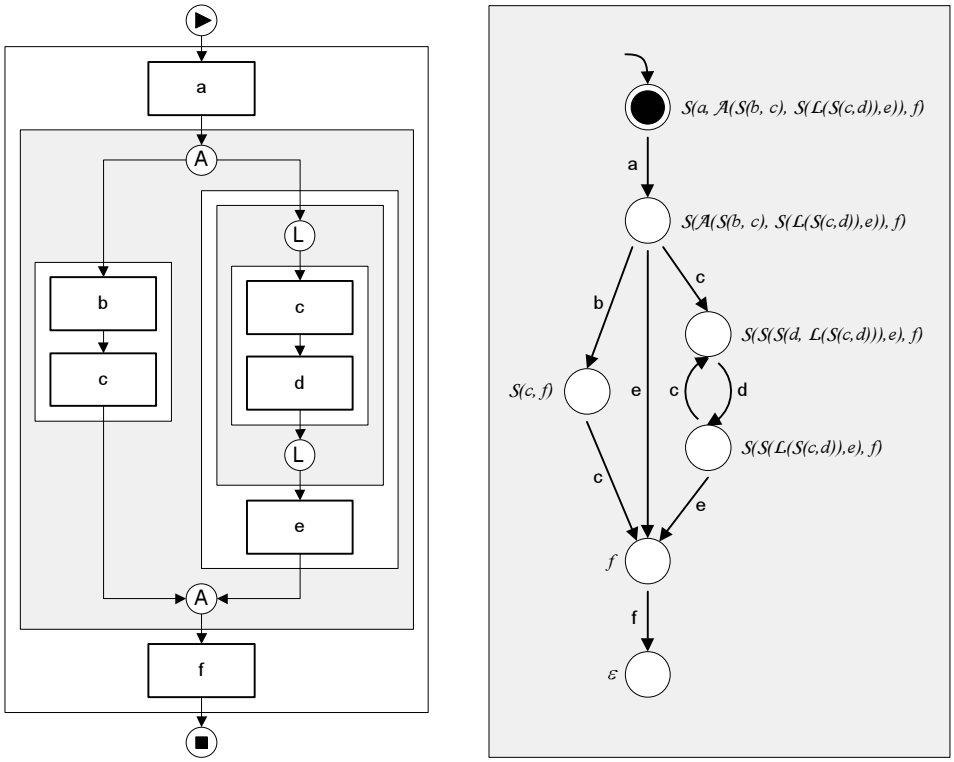


Abbildung 7.16: Beispiel für strukturierte operationelle Semantik linearer Prozessterme

7.4.3 Formale Semantik von CDL-Verhaltensbeschreibungen

Die Semantik der syntaktischen CDL-Konstrukte für die Beschreibung des passiven Verhaltens von Komponenten und Klassen (bzw. Kontrakten und Objekttypen) ergibt sich aus deren Abbildung auf die Metaklassen der UML-Protokollmaschinen, die wir im Rahmen der Dokumentation unseres Entwurfs in Abschnitt 9.1.2 im Detail vorstellen. Bei der Entwicklung eines formalen Modells für Verhaltensvergleiche, wie wir sie im Rahmen des Protokollvergleichs der geschäftsprozessorientierten Komponentensuche anstreben, ist allerdings die inhärent hohe Komplexität derartiger Vergleiche zu berücksichtigen. Um diese Komplexität zu begrenzen, verzichten wir in dem nachfolgend vorgestellten formalen Modell auf die Abbildung zweier Aspekte von CDL-Verhaltensbeschreibungen, die eben diese Komplexität „explodieren“ lassen können:

1. *Parametrisierung von Zuständen*: Die Parametrisierung der Zustände in CDL-Verhaltensbeschreibungen gestattet zwar die umfassende Beschreibung der Geschäftslogik einer Komponente, führt allerdings bei Verwendung von Parametern mit unendlichen Domänen zur Entstehung unendlicher Zustandsräume. In unserer Arbeit zur Komponentensuche verzichten wir vollständig auf die Berücksichtigung von Zustandsparametern und stellen auf diese Weise sicher, dass nur endliche Zustandsräume zu betrachten sind.[3] Dazu ist allerdings anzumerken, dass das im Protokollvergleich betrachtete Verhalten aufgrund dieser rigorosen Beschränkung von dem mittels CDL tatsächlich spezifizierten Verhalten abweicht. Unser Vorgehen lässt sich jedoch damit rechtfertigen, dass CDL-Verhaltensbeschreibungen grundsätzlich eine gewisse Unschärfe innewohnt. Insbesondere lässt sich der tatsächliche Zustand eines Geschäftsobjekts nicht aus seiner CDL-Beschreibung ableiten, wenn Referenzen auf dieses Objekt mit dem `existing`-Operator erzeugt wurden.

2. *Auswertung von Bedingungen*: Bedingungen in Call Events dienen der Beschreibung wichtiger Elemente der Geschäftslogik einer Komponente, ihre Auswertung zur Bestimmung der Ausführbarkeit der mit dem Call Event angesprochenen Operation kann allerdings außerordentlich komplex sein. Zwar lässt CDL in der vorliegenden Fassung nur einfache Vergleiche zwischen Ausdrücken zu (vgl. Anhang A), deren Auswertung bei weitem nicht die Komplexität allgemeiner prädikatenlogischer Formeln und die mit ihnen verbundene Entscheidbarkeitsproblematik aufweist, eine künftige Erweiterung ihrer Ausdrucksmächtigkeit erscheint jedoch durchaus denkbar und sinnvoll.

   Ein grundsätzlicheres Problem der Einbeziehung von Bedingungen in den Verhaltensvergleich besteht zudem darin, dass dies wiederum die Betrachtung der Parametrisierung von Zuständen voraussetzen würde, da Bedingungen zur Beschreibung der Ausführbarkeit von Operationen häufig in Abhängigkeit der Werte von Zustandsparametern formuliert werden.

Die eingeschränkte Betrachtung des Verhaltens von Komponenten und Klassen erlaubt es uns, auf den Einsatz des komplexen Modells der UML-Protokollmaschinen beim Protokollvergleich zu verzichten und wiederum die einfacheren Transitionssysteme als formale Grundlage der operationellen Verhaltensbeschreibung zu verwenden. Wir ordnen dazu jeder Komponente und jeder Klasse ein Transitionssystem zu, dessen Struktur sich aus der Abbildung der CDL-Verhaltensbeschreibungen auf die Metaklassen der UML-Protokollmaschinen ergibt (siehe Abschnitt 9.1.2).

Die Referenzierung neu erzeugter bzw. bereits existierender Instanzen von Klassen durch Operationen berücksichtigen wir, indem wir die jeweiligen Transitionssysteme durch eine spezielle Verweisrelation miteinander verknüpfen:

Definition 7.7 (Verweisrelation, Verweis) *Eine* Verweisrelation *zwischen Transitionssystemen* $\mathcal{T}_i = (A_i, Q_i, \rightarrow_i, q_{0_i}), (i = 1, 2)$ *ordnet Tupeln aus einem Transitionssystem und einer Aktion Tripel zu, die aus einem durch die Aktion referenzierten Tran-*

[3] Abschnitt 7.6 geht in aller Kürze auf die Technik der *Datenabstraktion* ein, mit der die effiziente Verarbeitung unendlicher Zustandsmengen unterstützt wird.

sitionssystem, der Anzahl referenzierter Instanzen dieses Transitionssystems sowie der Menge aktivierter Zustände bestehen:

$$\dashrightarrow \subseteq (LTS \times \Sigma_\tau) \times (LTS \times \mathbb{N} \times 2^{\Theta})$$

Ein Element $(\mathcal{T}_1, a, \mathcal{T}_2, n, P) \in \dashrightarrow$ *mit* $a \in A_1$ *und* $P \subseteq Q_2$ *heißt* Verweis *und wird als* $\mathcal{T}_1 \overset{a,n,P}{\dashrightarrow} \mathcal{T}_2$ *notiert. Es spezifiziert, dass die Ausführung der Aktion a im Transitionssystem* $\mathcal{T}_1$ *zur Erzeugung von n Instanzen des Transitionssystems* $\mathcal{T}_2$ *führt, wobei nicht näher spezifiziert ist, welcher der Zustände aus P aktiv ist.*

Wir setzen Verweise zur Beschreibung der Semantik der CDL-Operatoren `new` und `existing` ein. Wird eine Referenz auf ein durch das Transitionssystem $\mathcal{T}_2$ beschriebenes Geschäftsobjekt mit dem CDL-Operator `new` erzeugt, so befindet sich dieses Objekt nach Definition von `new` in seinem Anfangszustand, d. h. $P = \{q_{0_2}\}$. Wird ein solches Objekt über den `existing`-Operator referenziert, so kann es sich in jedem Zustand des Transitionssystems befinden, d. h. $P = Q_2$. Eine feinere Differenzierung der Mengen aktivierter Zustände, z. B. durch explizite Aufzählung potentieller Zustände des referenzierten Geschäftsobjekts, unterstützt CDL in ihrer aktuellen Fassung nicht.

Das Verhalten einer Komponente lässt sich nun wie folgt als komplexes Transitionssystem beschreiben:

Definition 7.8 (Komplexes Transitionssystem) *Ein* komplexes Transitionssystem *ist ein Tupel* $\mathcal{T}_\mathcal{X} = (T, A, Q, \dashrightarrow, \mathcal{T}_0)$*, wobei*

- $T = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ *eine Menge einfacher Transitionssysteme* $\mathcal{T}_i = (A_i, Q_i, \rightarrow_i, q_{0_i})$, $i \in \{1, \ldots, n\}$ *mit paarweise disjunkten Zustandsmengen* Q_i*,*
- $A := A_1 \cup \cdots \cup A_n$ *das Alphabet von* $\mathcal{T}_\mathcal{X}$*,*
- $Q := Q_1 \dot\cup \cdots \dot\cup Q_n$ *die Zustandsmenge von* $\mathcal{T}_\mathcal{X}$*,*
- $\dashrightarrow \subseteq (T \times A) \times (T \times \mathbb{N} \times 2^Q)$ *eine Verweisrelation und*
- $\mathcal{T}_0 = T_i \in T$, $i \in \{1, \ldots, n\}$ *das Starttransitionssystem von* $\mathcal{T}_\mathcal{X}$ *ist.*

Dabei repräsentieren die in T enthaltenen Transitionssysteme Verhaltensbeschreibungen der Komponente sowie der von ihr referenzierten Klassen und Objekttypen. Wir identifizieren einfache Transitionssysteme $\mathcal{T}_i$ *mit dem komplexen Transitionssystem* $\mathcal{T}_\mathcal{X}$*, das nur das einfache Transitionssystem enthält (*$\mathcal{T}_\mathcal{X} = (\{\mathcal{T}_i\}, A_i, Q_i, \varnothing, \mathcal{T}_i)$*).*

Das Transitionsverhalten eines komplexen Transitionssystems ist durch das Transitionsverhalten der umfassten einfachen Transitionssysteme sowie die Verweisrelation gegeben. Initial ist nur das Starttransitionssystem $\mathcal{T}_0$ des komplexen Transitionssystems aktiv, d. h. zur Ausführung von Aktionen bereit. Die Verweisrelation spezifiziert die Aktionen, bei deren Ausführung neue Instanzen der beteiligten Transitionssysteme aktiviert werden, die dann parallel zu dem initial aktiven Transitionssystem zur Ausführung von Aktionen bereit sind. Aufgrund dieses Parallelismus erhalten wir einen erweiterten Zustandsbegriff für komplexe Transitionssysteme:

Definition 7.9 (Komplexer Zustand) *Sei* $\mathcal{T}_\mathcal{X} = (T, A, Q, \dashrightarrow, \mathcal{T}_0)$ *ein komplexes Transitionssystem,* $T = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ *mit* $\mathcal{T}_i = (A_i, Q_i, \rightarrow_i, q_{0_i})$, $i \in \{1, \ldots, n\}$ *die Menge einfacher Transitionssysteme von* $\mathcal{T}_\mathcal{X}$ *und* $P_1, \ldots, P_m \subseteq Q$ *Mengen einfacher Zustände. Dann gilt:*

$$(P_1, \ldots, P_m) \textit{ ist ein } \text{komplexer Zustand } \textit{von } \mathcal{T}_\mathcal{X} \\ :\Leftrightarrow \quad \forall j \in \{1, \ldots, m\}\ \exists i \in \{1, \ldots, n\} \bullet P_j \subseteq Q_i$$

Wir schreiben $P_j \in c$ *für* $c = (P_1, \ldots, P_m)$, $j \in \{1, \ldots, m\}$, *um auszudrücken, dass* P_j *die Menge alternativ aktiver Zustände einer Transitionssysteminstanz des komplexen Zustands* c *repräsentiert. Wir identifizieren einfache Zustände* q *mit dem komplexen Zustand* $(\{q\})$ *und bezeichnen die Menge aller komplexen Zustände von* $\mathcal{T}_\mathcal{X}$ *mit* $\Pi_{\mathcal{T}_\mathcal{X}}$.

Ein komplexer Zustand repräsentiert folglich eine Struktur, die aus Mengen P_j einfacher Zustände gleichzeitig aktiver Transitionssysteme $\mathcal{T}_i \in T$ $(i \in \{1, \ldots, n\})$ besteht. Um die Transitionsrelation über komplexen Zuständen definieren zu können, benötigen wir noch Definitionen für die Äquivalenz sowie die Vereinigung komplexer Zustände:

Definition 7.10 (Äquivalenz und Vereinigung komplexer Zustände) *Seien* c, c', $d \in \Pi_{\mathcal{T}_\mathcal{X}}$ *komplexe Zustände mit* $c = (P_1, \ldots, P_n)$ *und* $d = (P'_1, \ldots, P'_m)$. *Dann gilt:*

- $c = d \quad :\Leftrightarrow \quad n = m \wedge (P \in c \Leftrightarrow P \in d)$ *(Äquivalenz)*
- $c' = c \cup d \quad :\Leftrightarrow \quad c' = (P_1, \ldots, P_n, P'_1, \ldots, P'_m)$ *(Vereinigung)*

Da die Reihenfolge der Zustandsmengen eines komplexen Zustands nicht relevant ist, sind zwei komplexe Zustände gleich, wenn sie gleich groß sind und die gleichen Zustandsmengen enthalten. Die Vereinigung zweier komplexer Zustände kann daher erreicht werden, indem die Zustandsmengen eines komplexen Zustands um die Zustandsmengen eines zweiten komplexen Zustands ergänzt werden.

Mit Hilfe des Wissens über komplexe Zustände können wir nun die Transitionsrelation komplexer Transitionssysteme definieren, die wir zunächst jedoch graphisch veranschaulichen wollen. In Abbildung 7.14 und 7.15 hatten wir bereits das Verhalten einfacher Transitionssysteme in den Fällen beschrieben, in denen die Ausführung einer Aktion nicht zur Referenzierung von Geschäftsobjekten und damit neuer Instanzen einfacher Transitionssysteme führt. Neben diesen Fällen, die analog für komplexe Transitionssysteme gelten, können wir drei weitere Fälle unterscheiden, bei denen komplexe Zustände durch die Erzeugung neuer Instanzen einfacher Transitionssysteme erweitert werden. Abbildung 7.17 beschreibt den Fall, in dem eine sichtbare Aktion ausgeführt wird, die eine Referenz auf ein neues Geschäftsobjekt erzeugt. Ausgehend von einer Konfiguration, in der nur eine Instanz des Transitionssystems A_1 mit aktivem einfachem Zustand $q_{1,0}$ bzw. komplexem Zustand $(\{q_{1,0}\})$ existiert, führt die Transition $\stackrel{a}{\Longrightarrow}$ aufgrund des Verweises $A_1 \stackrel{a,1,\{q_{2,0}\}}{\dashrightarrow} A_2$ zu einer Folgekonfiguration, in der nunmehr der einfache Folgezustand $q_{1,1}$ in A_1 aktiv ist und außerdem eine Instanz des Transitionssystems A_2 mit aktivem Zustand $q_{2,0}$ erzeugt wurde. Der resultierende komplexe Zustand ist damit durch das Tupel $(\{q_{1,1}\}, \{q_{2,0}\})$ gegeben, und die Transition kann als $(\{q_{1,0}\}) \stackrel{a}{\Longrightarrow} (\{q_{1,1}\}, \{q_{2,0}\})$ spezifiziert werden.

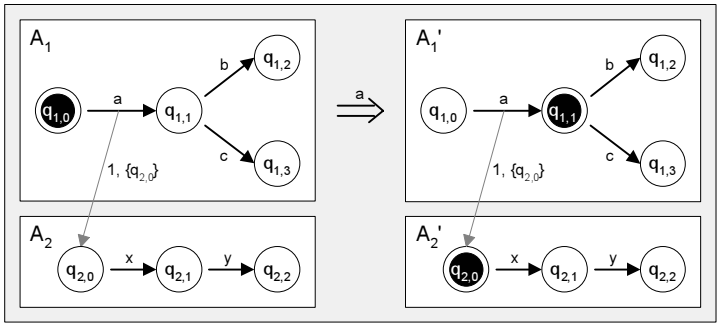


Abbildung 7.17: Direkte Ausführung einer Aktion mit Erzeugung einer neuen Referenz

Der zweite und dritte Fall behandeln den Umgang mit unsichtbaren Aktionen. In dem in Abbildung 7.18 dargestellten Fall kann die gewünschte Aktion b ausgeführt werden, indem zuvor eine unsichtbare Aktion ausgeführt wird. Diese τ-Aktion erzeugt aufgrund eines Verweises $A_1 \overset{\tau,1,\{q_{2,0}\}}{\dashrightarrow} A_2$ eine neue Instanz des Transitionssystems A_2. Die Anfangskonfiguration dieses Beispiels ähnelt der aus Abbildung 7.17, wobei nun jedoch die Aktion a in A_1 durch eine τ-Aktion ersetzt wurde. Die Folgekonfiguration nach Ausführung der Transition $\overset{b}{\Longrightarrow}$ enthält die bereits aktive Instanz von A_1 mit dem aktiven Zustand $q_{1,2}$ sowie eine neue Instanz des Transitionssystems A_2, bei der der Anfangszustand $q_{2,0}$ aktiv ist. Wir spezifizieren diese Transition als $(\{q_{1,0}\}) \overset{b}{\Longrightarrow} (\{q_{1,2}\}, \{q_{2,0}\})$.

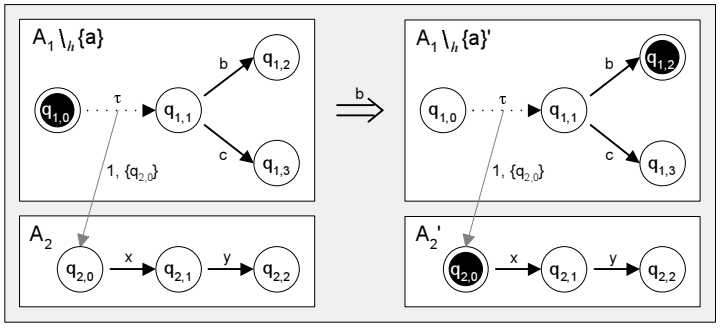


Abbildung 7.18: Indirekte Ausführung einer Aktion mit Erzeugung einer neuen Referenz

Die in Abbildung 7.19 dargestellte Situation betrachtet schließlich den Fall, bei dem eine geforderte Aktion durch eine neue Instanz eines Transitionssystems ausgeführt werden kann. In der abgebildeten Ausgangskonfiguration ist das Transitionssystem A_1 nicht in der Lage, die Aktion x zu bedienen. Allerdings wird nach Ausführung der versteckten Aktion im Zustand $q_{1,0}$ aufgrund des Verweises $A_1 \overset{\tau,1,\{q_{2,0}\}}{\dashrightarrow} A_2$ eine neue Instanz des Transitionssystems A_2 erzeugt, die die Ausführung der x-Aktion gestattet und danach von ihrem Startzustand $q_{2,0}$ in den Folgezustand $q_{2,1}$ übergeht. Die komplexe Transition lässt sich als $(\{q_{1,0}\}) \overset{x}{\Longrightarrow} (\{q_{1,1}\}, \{q_{2,1}\})$ spezifizieren.

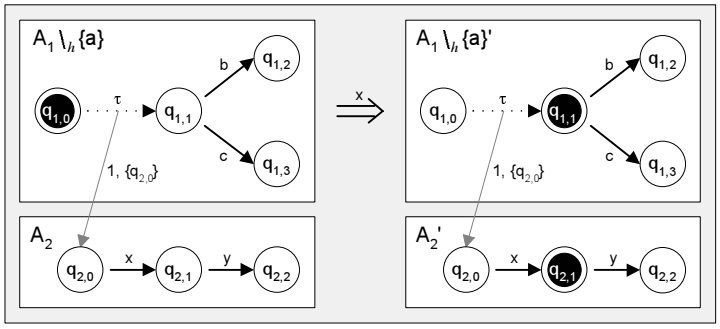


Abbildung 7.19: Indirekte Ausführung einer Aktion über eine neu erzeugte Referenz

Mit dieser Darstellung der zu berücksichtigenden Fälle können wir nun die Transitionsrelation formal definieren:

Definition 7.11 (Transitionsrelation komplexer Transitionssysteme) *Sei* $\mathcal{T}_\mathcal{X} = (T, A, Q, \dashrightarrow, \mathcal{T}_0)$ *ein komplexes Transitionssystem,* $T = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ *mit* $\mathcal{T}_i = (A_i, Q_i, \rightarrow_i, q_{0_i})$, $i \in \{1, \ldots, n\}$ *die Menge einfacher Transitionssysteme von* $\mathcal{T}_\mathcal{X}$ *und* $c = (P_1, \ldots, P_m)$ *ein komplexer Zustand von* $\mathcal{T}_\mathcal{X}$*. Sei ferner* c' *ein weiterer komplexer Zustand. Dann gilt:*

$$
\begin{aligned}
& c \overset{a}{\Longrightarrow} c' \\
:\Leftrightarrow\quad & \exists_{1\le i\le n,\, 1\le j\le m}\ p \in P_j \subseteq Q_i,\ q \in Q_i \bullet \\
& \quad (\quad p \overset{a}{\longrightarrow}_i q \\
& \quad \wedge\ c' = (P_1, \ldots, P_{j-1}, \{q\}, P_{j+1}, \ldots, P_m) \bigcup_{\mathcal{T}_i \overset{a,l,P}{\dashrightarrow} \mathcal{T}_k} \underbrace{(P, \ldots, P)}_{l\text{-}mal}\ ) \\
& \vee\ (\quad p \overset{\tau}{\longrightarrow}_i q \\
& \quad \wedge\ ((P_1, \ldots, P_{j-1}, \{q\}, P_{j+1}, \ldots, P_m) \bigcup_{\mathcal{T}_i \overset{\tau,l,P}{\dashrightarrow} \mathcal{T}_k} \underbrace{(P, \ldots, P)}_{l\text{-}mal}\ ) \overset{a}{\Longrightarrow} c'\ )
\end{aligned}
$$

Ein komplexer Zustand kann folglich in einen anderen komplexen Zustand übergehen, wenn entweder die gewünschte Aktion in einer seiner Zustandsmengen ausführbar ist (vgl. Abbildung 7.17) oder wenn durch Ausführung einer beliebigen Anzahl unsichtbarer Aktionen ein komplexer Zustand erreicht wird, in dem die gewünschte Aktion ausführbar ist (vgl. Abbildung 7.18 und 7.19).

In den Beispielen aus Abbildung 7.17-7.19 haben wir bislang nur Fälle betrachtet, in denen zum einen initial nur ein Zustand aktiv war und zum anderen nur Verweise $A_1 \overset{a,n,P}{\dashrightarrow} A_2$ existierten, die gemäß der Semantik des `new`-Operators Instanzen von Transitionssystemen erzeugt haben, bei denen lediglich der Startzustand aktiviert war ($|P| = 1$). Die Erzeugung von Referenzen auf Geschäftsobjekte mit dem `existing`-Operator wird formal durch Verweise modelliert, die mehrere Zustände einer neu erzeugten Transitionssysteminstanz aktivieren ($|P| > 1$). Damit wird ausgedrückt, dass sich das referenzierte Objekt in einem der Zustände aus P befindet, der exakte Zustand des Objekts allerdings unbekannt ist. Soll eine Aktion in einem solchen Transitionssystem ausgeführt werden, so

wird der Kreis dieser potentiellen Zustände durch die optimistische Annahme eingegrenzt, dass das Geschäftsobjekt in einem der Zustände aus P ist, die die Ausführung der Aktion gestatten. Damit reduziert sich die Menge aktiver Zustände nach Ausführung der Aktion auf den Folgezustand der gewählten Transition (vgl. auch Definition 7.11). Abbildung 7.20 zeigt ein Beispiel für dieses Transitionsverhalten anhand eines Transitionssystems A, bei dem zunächst die Zustände q_0, q_1 und q_2 aktiv sind. Da nur der Zustand q_0 die gewünschte Aktion a ermöglicht, werden die beiden übrigen Zustände nicht weiter betrachtet. Nach der Ausführung von a ist nur der Folgezustand von q_0 aktiv (dies ist abhängig von der gewählten Transition entweder der Zustand q_1 oder q_2).

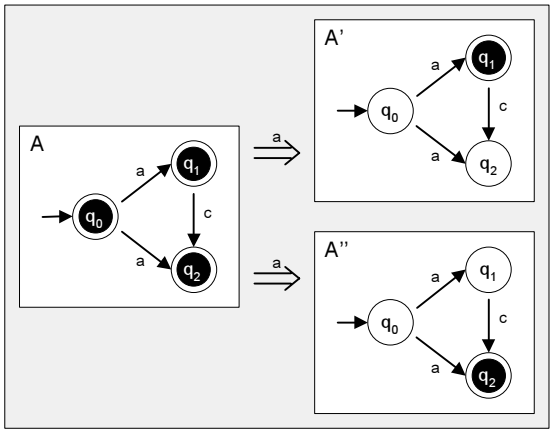


Abbildung 7.20: Ausführung einer Aktion bei mehreren aktiven Zuständen

Neben der Illustration des Transitionsverhaltens komplexer Zustände mit mehreren aktiven Zuständen zeigt dieses Beispiel auch den Umgang der Transitionsrelation mit Nichtdeterminismus. Im Zustand q_0 liegt bzgl. der Aktion a eine nichtdeterministische Entscheidung vor. Anstatt eine Transition $(\{q_0, q_1, q_2\}) \stackrel{a}{\Longrightarrow} (\{q_1, q_2\})$ zu erlauben, gestattet die Transitionsrelation lediglich zwei alternative Transitionen $(\{q_0, q_1, q_2\}) \stackrel{a}{\Longrightarrow} (\{q_1\})$ und $(\{q_0, q_1, q_2\}) \stackrel{a}{\Longrightarrow} (\{q_2\})$. Damit wird die Menge der in einer Instanz eines Transitionssystems potentiell aktiven Zustände frühzeitig reduziert.

7.5 Eine Behavioural-Subtyping-Relation auf dynamisch erweiterbaren Prozessen

Basierend auf den Anforderungen an eine Behavioural-Subtyping-Relation für die geschäftsprozessorientierte Komponentensuche aus Abschnitt 7.2, dem Wissen um bestehende Subtyping-Relationen aus Abschnitt 7.3 und den formalen Grundlagen aus dem vorangegangenen Abschnitt 7.4 können wir nun unsere Behavioural-Subtyping-Relation *Match* ($\sqsubseteq_{match}$) vorstellen. In diesem Abschnitt führen wir zunächst die formale Definition von Match ein, bevor wir einen einfachen Algorithmus zur Berechnung dieser Relation angeben.

7.5.1 Formale Definition

Match leitet sich aus Weak Simulation ab und verlangt, dass eine Komponente in der Lage ist, Teile des in einem Geschäftsprozessmodell spezifizierten Verhaltens durch die bedarfsgerechte Bereitstellung geeigneter Operationen zu unterstützen. Da wir unseren Ansatz zur Bestimmung der Eignung einer Operation für die Unterstützung einer Aktivität erst in Kapitel 8 vorstellen, behelfen wir uns hier zunächst mit der Definition einer abstrakten Relation für die Zuordnung von Operationen zu Aktivitäten.

Definition 7.12 (Implementierungsrelation (abstrakt)) *Seien* $\mathcal{C} = (T_\mathcal{C}, A_\mathcal{C}, Q_\mathcal{C}, \dashrightarrow_\mathcal{C}, \mathcal{T}_{0_\mathcal{C}})$ *und* $\mathcal{R} = (T_\mathcal{R}, A_\mathcal{R}, Q_\mathcal{R}, \dashrightarrow_\mathcal{R}, \mathcal{T}_{0_\mathcal{R}})$ *komplexe Transitionssysteme zur Repräsentation des Verhaltens einer Komponente* $\mathcal{C}$ *und eines Geschäftsprozessmodells* $\mathcal{R}$*. Sei ferner* $o \in A_\mathcal{C}$ *eine Operation von* $\mathcal{C}$ *und* $a \in A_\mathcal{R}$ *eine Aktivität aus* $\mathcal{R}$*. Die Relation* Impl *ordnet Aktionen (Aktivitäten) geeignete Implementierungen in Gestalt anderer Aktionen (Operationen) zu:*

$$Impl \subseteq \Sigma \times \Sigma$$

Ein Element $(a, o) \in Impl$ *besagt, dass die Aktion* o *eine geeignete Implementierung der Aktion* a *darstellt („*o *implementiert* a*“). Wir schreiben alternativ auch* $o \in Impl(a)$*.*

Den Kern der Behavioural-Subtyping-Relation Match stellt die Relation *Embedded Weak Simulation* ($\sqsubseteq_{sim_{e\tau}}$) dar (vgl. auch [Kei01]). Embedded Weak Simulation unterscheidet sich von Weak Simulation dadurch, dass ein Subtyp nicht bereits ab dessen Startzustand das Verhalten des Supertyps simulieren können muss:

Definition 7.13 (Embedded Weak Simulation) *Seien* $\mathcal{R} = (T_\mathcal{R}, A_\mathcal{R}, Q_\mathcal{R}, \dashrightarrow_\mathcal{R}, \mathcal{T}_{0_\mathcal{R}})$ *und* $\mathcal{C} = (T_\mathcal{C}, A_\mathcal{C}, Q_\mathcal{C}, \dashrightarrow_\mathcal{C}, \mathcal{T}_{0_\mathcal{C}})$ *komplexe Transitionssysteme mit* $\mathcal{T}_{0_\mathcal{R}} = (A_{\mathcal{T}_{0_\mathcal{R}}}, Q_{\mathcal{T}_{0_\mathcal{R}}}, \rightarrow_{\mathcal{T}_{0_\mathcal{R}}}, q_{0_{\mathcal{T}_{0_\mathcal{R}}}})$ *und* $\mathcal{T}_{0_\mathcal{C}} = (A_{\mathcal{T}_{0_\mathcal{C}}}, Q_{\mathcal{T}_{0_\mathcal{C}}}, \rightarrow_{\mathcal{T}_{0_\mathcal{C}}}, q_{0_{\mathcal{T}_{0_\mathcal{C}}}})$*. Seien ferner* $c_\mathcal{R} \in \Pi_\mathcal{R}$ *und* $c_\mathcal{C} \in \Pi_\mathcal{C}$ *zwei komplexe Zustände. Dann gilt:*

$$\begin{aligned}
\mathcal{R} \sqsubseteq_{sim_{e\tau}} \mathcal{C} \quad :\Leftrightarrow \quad & \textit{es existiert eine binäre Relation } \sim_{e\tau} \subseteq \Pi_\mathcal{R} \times \Pi_\mathcal{C} \\
& \textit{und ein Zustand } q'_{0_{\mathcal{T}_{0_\mathcal{C}}}} \in Q_{\mathcal{T}_{0_\mathcal{C}}}\textit{, so dass} \\
& 1.\ (\{q_{0_{\mathcal{T}_{0_\mathcal{R}}}}\}) \sim_{e\tau} (\{q'_{0_{\mathcal{T}_{0_\mathcal{C}}}}\}) \\
& 2.\ c_\mathcal{R} \sim_{e\tau} c_\mathcal{C} \textit{ und } c_\mathcal{R} \stackrel{a}{\Longrightarrow} c'_\mathcal{R} \\
& \qquad \Rightarrow \exists\, o \in Impl(a),\ c'_\mathcal{C} \in \Pi_\mathcal{C} \bullet c_\mathcal{C} \stackrel{o}{\Longrightarrow} c'_\mathcal{C} \textit{ und } c'_\mathcal{R} \sim_{e\tau} c'_\mathcal{C}.
\end{aligned}$$

Embedded Weak Simulation berücksichtigt damit direkt die Abweichung „Embedding“, nach der es möglich sein soll, dass eine Komponente zunächst eine Folge von Initialisierungsaktionen ausführt, bevor sie für die Unterstützung des geforderten Geschäftsprozesses bereit ist. Um die als Component Interference und Process Interference bezeichneten Abweichungen zu erfüllen, müssen wir Embedded Weak Simulation noch um die Möglichkeit erweitern, zusätzliche, durch die Komponente bzw. den Geschäftsprozess „eingestreute“ Aktionen zu verstecken. Wir wählen dazu den von Fischer und Wehrheim [FW00] beim Optimistic Subtyping eingesetzten *Hiding*-Operator:

Definition 7.14 (Hiding) *Sei $\mathcal{T} = (A, Q, \rightarrow, q_0)$ ein einfaches Transitionssystem und $N \subseteq \Sigma$ eine Menge von Aktionen. Dann ist das Verstecken (Hiding) von N in $\mathcal{T}$ definiert als:*

$$\mathcal{T}\backslash_h N := (A\backslash N, Q, \rightarrow', q_0)$$

mit

$$\rightarrow' = \{(q, a, q') \in \rightarrow \mid a \notin N\} \cup \{(q, \tau_a, q') \mid \exists\, a \in N : (q, a, q') \in \rightarrow\}.$$

Sei des Weiteren $\mathcal{T}_\mathcal{X} = (T, A, Q, \dashrightarrow, \mathcal{T}_0)$ ein komplexes Transitionssystem. Hiding von N in $\mathcal{T}_\mathcal{X}$ ist wie folgt definiert:

$$\mathcal{T}_\mathcal{X}\backslash_h N := (T', A\backslash N, Q, \dashrightarrow', \mathcal{T}_0\backslash_h N)$$

mit

$$T' = \{\mathcal{T}_i' \mid \mathcal{T}_i \in T \wedge \mathcal{T}_i' = \mathcal{T}_i\backslash_h N\}$$

und

$$\begin{aligned}\dashrightarrow' = \quad & \{(\mathcal{T}_i', a, \mathcal{T}_j', n, P) \mid (\mathcal{T}_i, a, \mathcal{T}_j, n, P) \in \dashrightarrow \wedge\, a \notin N\} \\ \cup\, & \{(\mathcal{T}_i', \tau_a, \mathcal{T}_j', n, P) \mid (\mathcal{T}_i, a, \mathcal{T}_j, n, P) \in \dashrightarrow \wedge\, a \in N\}.\end{aligned}$$

Die Anwendung des Hiding-Operators auf ein (komplexes) Transitionssystem bewirkt, dass die zu versteckenden Aktionen durch unsichtbare Aktionen ersetzt werden. Wir indizieren dabei die resultierenden τ-Aktionen mit dem Namen der jeweils versteckten Aktion, um die Semantik der versteckten Aktionen durch eine entsprechend angepasste Verweisrelation in der Transitionsrelation korrekt berücksichtigen zu können.

Zustände, in denen unsichtbare Aktionen ausgeführt werden können, werden in Prozessalgebren allgemein als *instabil* bezeichnet, da der betrachtete Prozess in diesen Zuständen unvermittelt den Zustand wechseln kann. Da bei Subtyping-Relationen auf Basis von Failures Refinement im Allgemeinen nur das Verhalten in *stabilen* Zuständen betrachtet wird, führt der Einsatz des Hiding-Operators beim Optimistic Subtyping dazu, dass sichtbare Aktionen in instabilen Zuständen nicht mehr als gültiges Verhalten im Verhaltensvergleich berücksichtigt werden (vgl. Abschnitt 7.3.3 sowie das Beispiel in Abbildung 7.12). Aufgrund der Definition unserer Transitionsrelation $\Rightarrow$ (vgl. Definition 7.5) führt die Anwendung des Hiding-Operators auf (komplexe) Transitionssysteme in unserem Ansatz nicht zur Einführung instabiler Zustände: Jegliche Transition $p \stackrel{a}{\Longrightarrow} q$ ist von der Form $p \xrightarrow{\tau} \ldots \xrightarrow{\tau} p' \xrightarrow{a} q$, d. h. eine $\Rightarrow$-Transition wird nach Ausführung einer beliebigen Folge unsichtbarer τ-Aktionen immer mit einer sichtbaren Aktion abgeschlossen. Da unbemerkte Zustandswechsel aufgrund von Instabilität somit ausgeschlossen sind, repräsentieren Zustände mit ausgehenden τ-Transitionen weiterhin gültiges Verhalten einer Komponente bzw. eines Geschäftsprozessmodells.

Mit diesen Überlegungen sind wir nun in der Lage, unsere Subtyping-Relation *Match* zur Bestimmung der Übereinstimmung einer Komponente mit einem Geschäftsprozessmodell auf der Protokollebene zu definieren:

Definition 7.15 (Match) *Seien $\mathcal{R} = (T_\mathcal{R}, A_\mathcal{R}, Q_\mathcal{R}, \dashrightarrow_\mathcal{R}, \mathcal{T}_{0_\mathcal{R}})$ und $\mathcal{C} = (T_\mathcal{C}, A_\mathcal{C}, Q_\mathcal{C}, \dashrightarrow_\mathcal{C}, \mathcal{T}_{0_\mathcal{C}})$ komplexe Transitionssysteme, die die formale Semantik eines Geschäftsprozessmodells bzw. einer Komponente beschreiben. Dann gilt:*

$$\begin{aligned}&\mathcal{C} \text{ unterstützt die Ausführung von } \mathcal{R}\, (\mathcal{R} \sqsubseteq_{match} \mathcal{C}) \\ :\Leftrightarrow\ & \mathcal{R}_\tau \sqsubseteq_{sim_{e\tau}} \mathcal{C}_\tau\end{aligned}$$

mit

$$\mathcal{R}_\tau = \mathcal{R}\backslash_h\{a \in A_\mathcal{R} \mid \nexists o \in A_\mathcal{C} \bullet o \in Impl(a)\}$$

und

$$\mathcal{C}_\tau = \mathcal{C}\backslash_h\{o \in A_\mathcal{C} \mid \nexists a \in A_\mathcal{R} \bullet o \in Impl(a)\}.$$

Diese Definition reflektiert die Protokollabweichungen „Component Interference“ und „Process Interference“ aus Abschnitt 7.2 dadurch, dass mittels Hiding zum einen diejenigen Aktionen eines Geschäftsprozessmodells $\mathcal{R}$ versteckt werden, für die seitens der Komponente $\mathcal{C}$ keine Unterstützung angeboten wird, und zum anderen diejenigen Aktionen der Komponente $\mathcal{C}$ versteckt werden, die durch das Geschäftsprozessmodell $\mathcal{R}$ nicht gefordert werden.

7.5.2 Algorithmus

Wir stellen nun einen Algorithmus zur Berechnung der Relation Match vor und diskutieren Fragestellungen im Zusammenhang mit der Terminierung sowie der Komplexität dieses Algorithmus.

Spezifikation des Algorithmus Ein Algorithmus zur Berechnung der Behavioural-Subtyping-Relation Match zwischen komplexen Transitionssystemen $\mathcal{R}$ und $\mathcal{C}$ kann grob in zwei Schritte unterteilt werden (vgl. Definition 7.15):

1. Anwendung des Hiding-Operators auf $\mathcal{R}$ und $\mathcal{C}$
2. Bestimmung der Relation Embedded Weak Simulation zwischen den resultierenden Transitionssystemen $\mathcal{R}'$ und $\mathcal{C}'$

In der folgenden Beschreibung eines einfachen Backtracking-Algorithmus zur Bestimmung von Match gehen wir nicht weiter auf die Anwendung des Hiding-Operators im ersten Schritt ein und konzentrieren uns stattdessen auf den zweiten Schritt. Dabei verzichten wir darauf, die für die Berücksichtigung von Embedding erforderliche Iteration über mögliche Anfangszustände der Komponente darzustellen (vgl. Punkt 1 von Definition 7.13). Wir beschreiben in Listing 7.1 lediglich den Kern des Algorithmus durch eine rekursive Prozedur `check()` zur Berechnung der Subtyping-Relation Weak Simulation, die als Argumente komplexe Transitionssysteme `r` und `c` zur Repräsentation des Verhaltens eines Geschäftsprozessmodells und einer Komponente sowie zu vergleichende komplexe Zustände `rState` und `cState` dieser Transitionssysteme erhält. `check()` liefert `true` zurück, wenn die Zustände `rState` und `cState` gemäß Weak Simulation in Beziehung stehen, d. h. das Transitionssystem `c` vom Zustand `cState` ausgehend das im Transitionssystem `r` vom Zustand `rState` aus mögliche Verhalten simulieren kann.

Der Algorithmus verwaltet eine globale (initial leere) Struktur besuchter Zustandspaare `simRelation`, in der vermerkt wird, ob zwischen den jeweiligen Zuständen eine Beziehung gemäß Weak Simulation besteht. Existiert bei einem Aufruf der Prozedur `check()` kein Eintrag für das aktuell betrachtete Zustandspaar (Zeile 2), werden zunächst die in den Zuständen `rState` und `cState` ausführbaren Aktionen ermittelt (Zeilen 3 und 4). Sofern mindestens eine Aktion im Zustand `rState` auszuführen ist, also noch kein

```
1 boolean check(ComplexLTS r, ComplexLTS c, State rState, State cState) {
2   if simRelation(rState, cState) not defined {
3     let actionsR = actions enabled in rState
4     let actionsC = actions enabled in cState
5     if actionsR not empty {
6       if actionsR ⊆_Impl actionsC {
7         for all 'act' in actionsR {
8           let statesR = successor states of rState with regard to 'act'
9           let statesC = successor states of cState with regard to 'act'
10          for all 'nextR' in statesR {
11            FOUND = false;
12            while not FOUND and statesC not empty {
13              let nextC = fittest state from statesC
14              remove 'nextC' from statesC
15              FOUND = check(r, c, nextR, nextC);
16            }
17            if not FOUND {
18              simRelation(rState, cState) = false
19              return false
20            }
21          }
22        }
23      }
24      else {
25        simRelation(rState, cState) = false
26        return false
27      }
28    }
29    simRelation(rState, cState) = true
30  }
31  return simRelation(rState, cState)
32 }
```

Listing 7.1: Algorithmus zur Berechnung von *Match* (Weak Simulation)

Endzustand des komplexen Transitionssystems `r` erreicht wurde (Zeile 5), wird geprüft, ob für jede auszuführende Aktion eine entsprechende Aktion im Zustand `cState` des komplexen Transitionssystems `c` zur Ausführung bereit ist (Zeile 6). Dabei findet die Relation *Impl* Berücksichtigung, die gemäß Definition 7.12 Aktivitäten eines Geschäftsprozessmodells geeignete Operationen einer Komponente zuordnet. Sind im Zustand `cState` keine geeigneten Aktionen zur Ausführung bereit, wird dies in die Relation `simRelation` eingetragen, und der Algorithmus bricht die Untersuchung des Suchzweigs mit dem Wert `false` ab (Zeilen 25 und 26). Anderenfalls werden in einer Schleife über den in `rState` ausführbaren Aktionen `act` (Zeile 7) zunächst Mengen möglicher Folgezustände von `rState` und `cState` bzgl. `act` ermittelt (Zeilen 8 und 9). Bei der Bestimmung der Folgezustände wird die Transitionsrelation $\Rightarrow$ aus Definition 7.11 eingesetzt. Anschließend wird in zwei weiteren geschachtelten Schleifen (Zeilen 10 und 12) für jeden Zustand `nextR` aus der Menge der Folgezustände von `rState` ein Folgezustand `nextC` von `cState` gesucht, der mit `nextR` gemäß Weak Simulation in Beziehung steht. Für den gesuchten Zustand muss also gelten, dass der rekursive Aufruf des Algorithmus `check(r, c, nextR, nextC)` den

booleschen Wert `true` zurückliefert (Zeile 15). Bei der Auswahl des Zustands `nextC` aus der Menge aller Folgezustände von `cState` werden nach frei definierbaren Strategien die am geeignetsten erscheinenden Zustände zuerst untersucht (Zeile 13). Eine solche Strategie könnte z. B. darin bestehen, zuerst die Zustände zu untersuchen, die die geringste Anzahl parallel aktivierter Transitionssysteme umfassen. Kann kein geeigneter Folgezustand `nextC` gefunden werden, so wird die Relation `simRelation` entsprechend ergänzt, und der Algorithmus bricht mit dem Wert `false` ab (Zeilen 17-19). Wird allerdings für jede ausführbare Aktion des Zustands `rState` eine entsprechende Aktion im Zustand `cState` derart gefunden, dass die resultierenden Folgezustände `nextR` und `nextC` gemäß Weak Simulation in Beziehung stehen, so wird dies in der Relation `simRelation` vermerkt und die Prozedur `check()` erfolgreich abgeschlossen (Zeilen 29 und 31).

Terminierung Die Terminierung des in Listing 7.1 vorgestellten Algorithmus hängt von zwei Bedingungen ab: Zum einen muss die Kette rekursiver Aufrufe der Prozedur `check()` in Zeile 15 abbrechen, und zum anderen dürfen in den Zeilen 8 und 9 keine unendlichen Zustandsmengen auftreten, die dann zu nicht terminierenden Schleifen in den Zeilen 10 und 12 führen. Unter der Voraussetzung endlicher Zustandsmengen wird die erste Bedingung durch die Verwaltung der bereits besuchten Zustandspaare in der Struktur `simRelation` und die darauf zurückgreifende Abbruchbedingung für die Rekursion erreicht (vgl. Zeilen 2, 18, 25 und 29 des Algorithmus). Die zweite Anforderung ist für die Menge `statesR` der Folgezustände des Zustands `rState` immer erfüllt, da ein Geschäftsprozessmodell bzw. das entsprechende komplexe Transitionssystem `r` mangels Verweisen nur eine endliche Anzahl von Zuständen aufweist. Im Falle einer Komponente bzw. des ihr Verhalten beschreibenden komplexen Transitionssystems `c` lassen sich jedoch leicht Beispiele mit einer unendlichen Menge `statesC` komplexer Folgezustände des Zustands `cState` finden. So hat der Zustand $(\{q_{1,0}\})$ der Komponente $\mathcal{C}$ aus Abbildung 7.21 bzgl. der Aktion a eine unendliche Menge von Folgezuständen $\{(\{q_{1,0}\},\{q_{2,1}\}), (\{q_{1,0}\},\{q_{2,0}\},\{q_{2,1}\}), (\{q_{1,0}\},\{q_{2,0}\},\{q_{2,0}\},\{q_{2,1}\}), \ldots\}$, die über Transitionen $\stackrel{a}{\Longrightarrow}$ der Form $\cdot \xrightarrow{\tau} \cdots \xrightarrow{\tau} \cdot \xrightarrow{a} \cdot$ erreicht werden können. Das zentrale Problem bei der Beschränkung auf endliche Zustandsräume besteht dabei darin zu verhindern, dass – wie im Beispiel – über versteckte Aktionen unkontrolliert neue Instanzen von Transitionssystemen erzeugt werden.

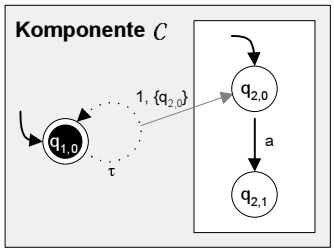


Abbildung 7.21: Beispiel für unendliche Menge von Folgezuständen

Abhilfe könnte geschaffen werden, indem der Algorithmus z. B. mit der maximalen Anzahl von Instanzen parametrisiert wird, die von einem Transitionssystem durch versteckte

Aktionen erzeugt werden dürfen. Wird ein solcher Parameter beispielsweise auf den Wert 2 festgesetzt, so ergibt sich als Folgezustandsmenge des Zustands ($\{q_{1,0}\}$) der Komponente $\mathcal{C}$ aus Abbildung 7.21 bzgl. der Aktion a nur noch $\{(\{q_{1,0}\}, \{q_{2,1}\}), (\{q_{1,0}\}, \{q_{2,0}\}, \{q_{2,1}\})\}$. Alternativ könnte auch verlangt werden, dass durch den Algorithmus generell nur die Erzeugung einer endlichen Anzahl von parallel aktiven Transitionssystemen zugelassen wird. Eine entsprechende Anforderung könnte in der Implementierung der Transitionsrelation $\Rightarrow$ berücksichtigt werden.

Komplexitätsbetrachtungen Der in Listing 7.1 vorgestellte Algorithmus ist relativ leicht nachzuvollziehen, hinsichtlich der Komplexität aber nicht optimal. Nach Hüttel und Shukla [HS96] sind die hier betrachteten Simulationsrelationen für endliche Transitionssysteme in polynomieller Zeit entscheidbar. Um einen Eindruck von der Komplexität des Algorithmus zu bekommen, unternehmen wir den Versuch einer *Worst-Case-Aufwandsabschätzung*. Seien dazu n_r und n_c die Anzahl der komplexen Zustände der komplexen Transitionssysteme `r` und `c`, auf die der Algorithmus angesetzt wird. Seien ferner $\alpha_{\mathcal{R}}$ und $\alpha_{\mathcal{C}}$ die Alphabete von `r` und `c`. Dann können wir den einzelnen Schritten des Algorithmus die in Listing 7.2 dargestellten Aufwände zuordnen.

```
1
2
3   n_r * |α_R| +                  /* max. Verzweigung eines Zustands in r  */
4   n_c * |α_C| +                  /* max. Verzweigung eines Zustands in c  */
5
6
7   |α_R| * (                      /* max. Anzahl in r ausführbarer Aktionen  */
8           n_r^2                  /* Tiefensuche mit quadratischem Aufwand  */
9           + n_c^2                /* Tiefensuche mit quadratischem Aufwand  */
10          + n_r * (              /* |statesR| ≤ n_r  */
11
12                  n_c * (        /* |statesC| ≤ n_c  */
13                                 /* Annahme: keine Bewertung der Zustände  */
14
15                                 /* ⇒ Gesamtanzahl von Aufrufen ≤ n_r * n_c  */
16                      )
17
18
19
20
21                  )
22          )
23
24  ...
```

Listing 7.2: Bewertung der Komplexität einzelner Schritte des Algorithmus

Als Aufwand für einen Durchlauf der Prozedur `check()` ohne Betrachtung rekursiver Aufrufe ergibt sich $n_r|\alpha_{\mathcal{R}}| + n_c|\alpha_{\mathcal{C}}| + |\alpha_{\mathcal{R}}|(n_r^2 + n_c^2 + n_r n_c)$. Da diese Prozedur maximal für alle Kombinationen von Zuständen `rState` und `cState`, also $(n_r n_c)$-mal aufgerufen werden muss (Zeile 15), erhalten wir als Gesamtaufwand für den Algorithmus

$n_r^2 n_c|\alpha_{\mathcal{R}}| + n_r n_c^2|\alpha_{\mathcal{C}}| + |\alpha_{\mathcal{R}}|(n_r^3 n_c + n_r n_c^3 + n_r^2 n_c^2)$. Wenn wir für die Größe der Alphabete $\alpha_{\mathcal{R}}$ und $\alpha_{\mathcal{C}}$ konstante Durchschnittswerte voraussetzen, können wir dem Algorithmus also polynomiellen Aufwand der Größenordnung $\mathcal{O}(n^4)$ für $n = \max(n_r, n_c)$ zuschreiben. Dieser Aufwand ist insbesondere dadurch problematisch, dass die Anzahl der komplexen Zustände n_c einer Komponente `c` aufgrund der Möglichkeit zur dynamischen Erzeugung von Transitionssystemen schnell hohe Werte annehmen kann. So hat das komplexe Transitionssystem aus Abbildung 7.21 grundsätzlich eine unendliche Anzahl von komplexen Zuständen. Beschränkt man sich wie oben vorgeschlagen beispielsweise darauf, durch versteckte Aktionen nur maximal zwei Instanzen eines Transitionssystems zu erzeugen, kommt man auf 7, bei drei Instanzen bereits auf 15 komplexe Zustände, die den anfallenden Berechnungsaufwand polynomiell beeinflussen.

7.6 Zusammenfassung

In diesem Kapitel haben wir zunächst eine Einführung in das Behavioural Subtyping und dessen Einsatz beim Protokollvergleich gegeben. Im Anschluss haben wir unsere initialen Anforderungen an den Protokollvergleich abgeschwächt, indem wir mit *Embedding*, *Component Interference* und *Process Interference* drei Formen zulässiger Protokollabweichungen definiert haben. Ausgehend von der resultierenden informellen Charakterisierung einer für die geschäftsprozessorientierte Komponentensuche geeigneten Behavioural-Subtyping-Relation haben wir verschiedene bekannte Relationen untersucht und hinsichtlich ihrer Eignung bewertet. Dabei hat sich gezeigt, dass *Weak Simulation* in Verbindung mit *Hiding* von ausgewählten Aktionen den Anforderungen am ehesten gerecht wird.

Da bislang lediglich eine informelle Definition der formalen Semantik linearer Prozessmodelle veröffentlicht ist, haben wir eine strukturierte operationelle Semantik formuliert, die einem linearen Prozessterm ein Transitionssystem zuordnet. Wie haben darüber hinaus auf Basis von Transitionssystemen ein formales Modell zur Beschreibung des Verhaltens von Komponenten entwickelt, das die dynamische Erzeugung von Geschäftsobjekten durch die Ausführung von Operationen berücksichtigt. Auf der Grundlage dieser Formalisierungen sowie der Relation Weak Simulation und dem Hiding-Operator haben wir schließlich unsere Behavioural-Subtyping-Relation *Match* entwickelt, die wir für den Protokollvergleich einsetzen. Wir haben zudem einen einfachen Algorithmus zur Berechnung von Match angegeben, den wir hinsichtlich Terminierung und Komplexität kurz diskutiert haben.

Sinnvolle Erweiterungen bzw. Verbesserungen des in diesem Kapitel vorgeschlagenen Ansatzes zum Protokollvergleich betreffen das betrachtete Verhalten von Komponenten sowie den Algorithmus zur Berechnung von Match. Bezüglich des betrachteten Verhaltens wäre eine Berücksichtigung der Parametrisierung von Zuständen wünschenswert, um ein präziseres Bild von der Eignung einer Komponente gewinnen zu können. Eine entsprechende Erweiterung würde allerdings voraussetzen, dass der CDL-Operator `existing` den Zustand (bzw. eine Menge möglicher Zustände) eines referenzierten Geschäftsobjekts spezifiziert. Arbeiten auf dem Gebiet der *Datenabstraktion* (vgl. z. B. [Weh99]) setzen sich in diesem Zusammenhang mit dem Problem der verhaltenserhaltenden Reduzierung großer Zustandsmengen auseinander, wie sie z. B. durch die Parametrisierung von Zuständen

entstehen können. Dabei kommen Abstraktionsfunktionen auf den Domänen der Parameter von Zuständen und Operationen zum Einsatz.

Im Hinblick auf den vorgeschlagenen Algorithmus erscheint eine Reduzierung der Komplexität vonnöten. Groote und Vaandrager stellen in [GV90] einen Algorithmus zur Bestimmung von Branching Bisimulationsbeziehungen zwischen Zuständen eines Transitionssystems vor. Dieser Algorithmus basiert auf einem effizienten Algorithmus zur Partitionierung von Transitionssystemen und erzeugt lediglich Aufwand der Größenordnung $\mathcal{O}(mn)$, wobei m die Anzahl der Kanten und n die Anzahl der Knoten des Transitionssystems angibt. Obwohl Branching Bisimulation eine Äquivalenzrelation und ihre asymmetrische Variante Branching Simulation schärfer als die von uns betrachtete Weak Simulation ist, legt die konzeptionelle Nähe dieser Relationen jedoch die Vermutung nahe, dass ein ähnlich effizienter Algorithmus für die Bestimmung von Beziehungen gemäß Weak Simulation existiert.

Kapitel 8

Geschäftsprozessorientierte Komponentensuche: Semantikvergleich

Neben dem Vergleich von Geschäftsprozessmodellen und Komponentenbeschreibungen auf der Protokollebene erfordert die Bestimmung der Eignung einer Komponente für die Unterstützung eines Geschäftsprozesses einen Vergleich auf der Semantikebene. In diesem Kapitel stellen wir entsprechende Konzepte zum Vergleich von Geschäftsprozessmodellen und Komponentenbeschreibungen vor. Die Einordnung des Semantikvergleichs als Aktivität der Suchphase im Makroprozess der geschäftsprozessorientierten Komponentensuche ist in Abbildung 8.1 dargestellt.

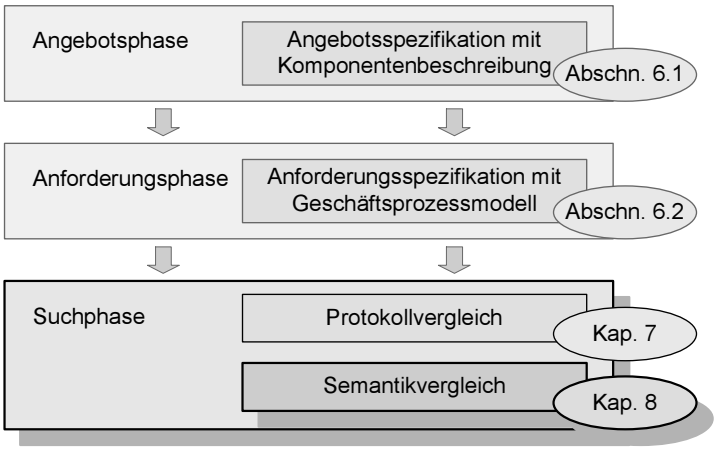


Abbildung 8.1: Einordnung des Kapitels in den Makroprozess

In Abschnitt 8.1 dieses Kapitels vergleichen wir zunächst Alternativen zur Spezifikation der fachlichen Semantik von Aktivitäten eines Geschäftsprozessmodells und Operationen einer Komponente. Ausgehend von einer Bewertung dieser Alternativen geben wir in Abschnitt 8.2 eine Einführung in den Ansatz der Normsprachen, bevor wir in

Abschnitt 8.3 eine normsprachliche Repräsentation der fachlichen Semantik von Aktivitäten und Operationen entwickeln und ein terminologisches Modell zur rechnergestützten Verwaltung einer entsprechenden Normsprache vorstellen. Abschnitt 8.4 führt zunächst formale Definitionen der fachlichen Semantik von Aktivitäten und Operationen auf Basis normsprachlicher Sätze ein und definiert anschließend auf dieser Grundlage, wann eine Operation einen für die Unterstützung der Ausführung einer Aktivität geeigneten Dienst implementiert. Abschnitt 8.5 schließt das Kapitel mit einer Zusammenfassung ab.

8.1 Alternativen für die Spezifikation fachlicher Semantik

In Kapitel 7 haben wir die *formale* Semantik von linearen Prozessmodellen und CDL-Komponentenbeschreibungen über ein abstraktes Maschinenmodell definiert. Diese Semantik weist aufgrund ihres formalen Charakters allerdings keinen Bezug zu dem konkreten Anwendungsgebiet der beschriebenen Geschäftsprozesse und Komponenten auf. Da dieser Anwendungsbezug jedoch für die geschäftsprozessorientierte Komponentensuche unabdingbar ist, ist die formale Semantik um eine *fachliche* Semantik zu ergänzen, die einem abstrakten Maschinenmodell durch Beschreibung der fachlichen Bezüge eine konkrete Interpretation zuordnet. Erst durch die Kombination von formaler und fachlicher Semantik erhalten wir eine umfassende Beschreibung der Bedeutung von Geschäftsprozessmodellen und Komponentenbeschreibungen.

Ähnlich wie beim zustandsbasierten Behavioural Subtyping beabsichtigen wir, die fachliche Semantik eines Geschäftsprozessmodells bzw. einer Komponentenbeschreibung zu definieren, indem wir den einzelnen Aktivitäten bzw. Operationen eine fachliche Semantik zuordnen. Hinsichtlich der Repräsentation dieser fachlichen Semantik sind allerdings verschiedene Ansätze denkbar, die im Folgenden diskutiert werden sollen.

8.1.1 Spezifikation der fachlichen Semantik von Operationen

Bei der Spezifikation der fachlichen Semantik von Operationen können wir grundsätzlich zwischen einem zustandsorientierten und einem vorgangsorientierten Ansatz unterscheiden. In der folgenden Erläuterung dieser Alternativen beziehen wir uns auf das in Abbildung 8.2 dargestellte Klassendiagramm aus dem Bereich der Auftragsverwaltung. Die Klasse `OrderManagement` verwaltet Aufträge in Form von `Order`-Objekten. Sie bietet die Möglichkeit, über die Operation `createOrder()` einen neuen Auftrag auf der Grundlage eines durch ein `Offer`-Objekt gegebenen Angebots anzulegen sowie über die Operation `delOrder()` einen bestehenden Auftrag zu löschen.

Beim *zustandsorientierten* Ansatz wird die Semantik von Operationen als Wirkung ihrer Ausführung über einem Zustandsraum spezifiziert. Dabei werden die einzelnen Zustände des Zustandsraums im Allgemeinen durch Zusicherungen über einer konkreten Implementierung der Schnittstelle definiert, der die Operation angehört (z. B. mittels der Object Constraint Language (OCL)). Die Semantik einer Operation wird dann wie in Abschnitt 3.4 vorgestellt als Korrektheitsformel durch Zusicherungen zur Spezifikation von Vor- und Nachbedingungen angegeben. Dieser Ansatz zur Repräsentation der Semantik von Operationen gestattet die Erstellung präziser Spezifikationen, die für die formale Verifikation von Eigenschaften herangezogen werden können (siehe auch [AO94]).

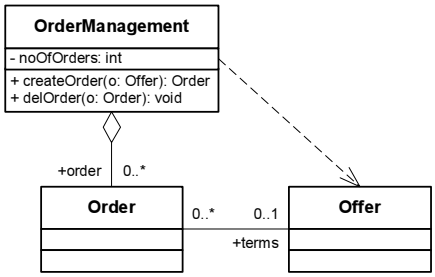


Abbildung 8.2: Einfaches Klassendiagramm zur Auftragsverwaltung

Das folgende Listing zeigt exemplarisch OCL-Spezifikationen der Semantik der Operationen `createOrder()` und `delOrder()` aus Abbildung 8.2:

```
context OrderManagement::createOrder(o: Offer): Order
  post:
    result.oclIsNew() and result.oclIsTypeOf(Order)
      and order = order@pre->including(result)
      and noOfOrders = noOfOrders@pre + 1
      and result.terms = o

context OrderManagement::delOrder(o: Order): void
  pre:
    order->includes(o)
  post:
    not order->includes(o)
      and noOfOrders = noOfOrders@pre - 1
```

Diese Spezifikationen drücken formal aus, dass nach Ausführung der Operation `createOrder()` ein neues Objekt der Klasse `Order` angelegt worden ist, das sich auf die im übergebenen `Offer`-Objekt festgelegten Bedingungen bezieht. Das neue `Order`-Objekt ist in die von dem angesprochenen `OrderManagement`-Objekt verwaltete Menge `order` von Auftragsobjekten aufgenommen worden. Zudem ist die Anzahl der verwalteten Objekte `noOfOrders` inkrementiert worden. Für die Operation `delOrder()` gilt die Vorbedingung, dass das zu löschende `Order`-Objekt in der Menge der verwalteten Auftragsobjekte enthalten sein muss. Nach Ausführung der Operation wird durch die Spezifikation zugesichert, dass das betreffende Objekte aus dieser Menge entfernt und die Anzahl der verwalteten Objekte entsprechend dekrementiert worden ist.

In dieser zustandsorientierten Spezifikation wird für die Formulierung der Vor- und Nachbedingungen auf Implementierungsdetails der Klassen `OrderManagement` und `Order` wie z. B. das Attribut `noOfOrders` und die Assoziation `order` zurückgegriffen. Diese (partielle) Veröffentlichung von Implementierungsdetails (z. B. in einer Komponentenbeschreibung) ist aus Sicht eines Komponentenherstellers kritisch zu bewerten, da dieser u. U. Wettbewerbsvorteile gefährdet sehen könnte. Für die Beschreibung von Kontrakten ist dieser Weg darüber hinaus im Allgemeinen nicht gangbar, da die vollständige Spezifikation der Semantik einer Operation vielfach den Bezug auf eine Implementierung zwingend voraussetzt. Abhilfe kann hier die Verwendung eines abstrakten Modells verschaf-

fen, das gewissermaßen als Referenzimplementierung eines zu beschreibenden Kontrakts oder einer Komponente dient, jedoch unabhängig von einer tatsächlich existierenden Implementierung ist (vgl. [LD00]). Anhand eines solchen abstrakten Modells lässt sich die Semantik von Operationen vollständig beschreiben. Unabhängig davon, ob ein abstraktes oder konkretes, d. h. auf eine existierende Implementierung bezogenes Modell zugrunde liegt, können Zusicherungen zur Formulierung von Vor- und Nachbedingungen schon bei relativ einfachen Operationen komplex werden (siehe z. B. [Ack01]). Diese für Fachexperten bereits aufgrund ihrer mathematischen, nicht operationellen Notation schwierig nachzuvollziehenden Zusicherungen erschweren damit die Bewertung von Komponenten, die im Rahmen der geschäftsprozessorientierten Komponentensuche gefunden werden.

Eine Alternative zur formalen zustandsorientierten Spezifikation besteht darin, die fachliche Semantik einer Operation explizit als (natürlich-)sprachliche Beschreibung des implementierten Dienstes anzugeben. Dieser *vorgangsorientierte* Ansatz geht von der Beobachtung aus, dass sich Anwendungssysteme als sprachliche Gebilde verstehen lassen, in denen Operationen als einfache (natürlich-)sprachliche Aussageschemata mit Subjekt, Prädikat, direktem Objekt und indirekten Objekten repräsentiert werden. Die Ausführung einer Operation entspricht dann einer *Aktualisierung*, d. h. einer Ausprägung eines solchen Aussageschemas in Form einer Aussage.

Die fachliche Semantik der Operation `createOrder()` der Klasse `OrderManagement` aus Abbildung 8.2 könnte beispielsweise durch das Aussageschema „OrderManagement erzeugt einen Auftrag auf Basis eines Angebots“ formuliert werden. Analog kann die Semantik der Operation `delOrder()` durch das Aussageschema „OrderManagement löscht einen Auftrag“ definiert werden. Der inhaltliche Aufbau beider Schemata ist eng mit der Signatur der jeweiligen Operation verknüpft. So haben wir die Stellung des Subjekts mit dem Besitzer der Operation, also der Klasse `OrderManagement`, belegt. Das Prädikat („erzeugen“ bzw. „löschen“) konnten wir aus dem Bezeichner der jeweiligen Operation ableiten. Bezüglich des direkten Objekts „Auftrag“ unserer Aussageschemata ist zu beachten, dass der Begriff „Order“ (bzw. das deutsche Pendant „Auftrag“) in den Signaturen beider Operationen zweifach auftritt: zum einen im Namen der Operation und zum anderen als Bezeichner des Typs eines erzeugten bzw. gelöschten Objekts. Damit wird in der Signatur der Operationen explizit zwischen einer *intensionalen* und einer *extensionalen* Sicht auf den Auftrag unterschieden. Während sich der Name der Operation auf das abstrakte Konzept (die *Intension*) bezieht, das durch den Begriff bezeichnet wird, bezieht sich das Objekt auf eine Instanz des Konzepts (ein Element der *Extension*). In unseren Aussageschemata haben wir diese differenzierte Betrachtung nicht wiedergegeben. Das indirekte Objekt „Angebot“ unseres ersten Aussageschemas bezieht sich schließlich auf den Parameter der Operation `createOrder()`.

Ähnlich wie formale Spezifikationen im zustandsorientierten Ansatz ein (konkretes oder abstraktes) *mathematisches* Modell des zu beschreibenden Weltausschnitts benötigen, erfordert die vorgangsorientierte Spezifikation ein *terminologisches* Modell, auf dessen Grundlage Aussageschemata entwickelt werden können. Ein solches terminologisches Modell ist durch eine Terminologie gegeben, die die Begriffswelt des betrachteten Anwendungsbereichs beschreibt. Obgleich der vorgangsorientierte Ansatz aufgrund seiner natürlichsprachlichen Basis nicht die Präzision des zustandsorientierten Ansatzes erreicht, bietet er den Vorteil, einfache, auch für Fachexperten nachvollziehbare Beschrei-

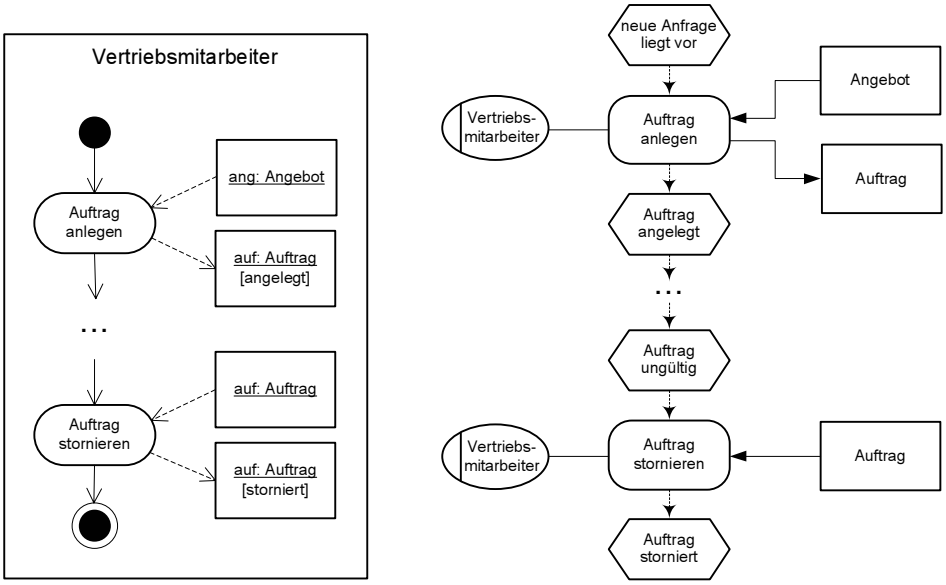
bungen der fachlichen Semantik von Operationen zu ermöglichen. Da es sich zudem bei einem terminologischen Modell immer um ein abstraktes Modell handelt, verhindert die vorgangsorientierte Spezifikation auch die von Komponentenherstellern kritisch gesehene Offenlegung von Implementierungsdetails. Als problematisch bei der Verwendung natürlichsprachlicher Spezifikationen ist neben der vergleichsweise niedrigen Präzision auch die Komplexität ihrer maschinellen Verarbeitung anzusehen.

8.1.2 Spezifikation der fachlichen Semantik von Aktivitäten

Aktuelle Sprachen zur Geschäftsprozessmodellierung definieren keine fachliche Semantik der modellierten Aktivitäten, sondern beschränken sich auf die Angabe einer mehr oder weniger vollständigen formalen Semantik. Dahinter liegt die Annahme, dass sich die inhaltliche Bedeutung der informell beschriebenen Aktivitäten dem Adressaten aufgrund der verwendeten Kombination graphischer Modellierungstechniken mit einfachen natürlichsprachlichen Ausdrücken mit hinreichender Präzision intuitiv erschließt.

Analog zur Spezifikation der fachlichen Semantik von Operationen kann grundsätzlich auch bei der Formulierung der fachlichen Semantik von Aktivitäten zwischen einem zustandsorientierten und einem vorgangsorientierten Ansatz unterschieden werden. Da sich Geschäftsprozessmodelle jedoch in erster Linie an Fachexperten richten und ihre maschinelle Verwertbarkeit erst in zweiter Linie interessiert, ist die zustandsorientierte Formulierung der fachlichen Semantik von Aktivitäten aufgrund der bereits in Abschnitt 8.1.1 genannten Nachteile nicht sinnvoll. Aktuelle Sprachen zur Geschäftsprozessmodellierung stellen folgerichtig die auszuführenden Aktivitäten und nicht die durch deren Ausführung manipulierten Daten in den Vordergrund der Betrachtung. Bezüge zur zustandsorientierten Spezifikation der fachlichen Semantik von Aktivitäten finden sich dennoch z. B. in Aktivitätsdiagrammen und EPKs. So ermöglichen Aktivitätsdiagramme die explizite Modellierung des Zustands von Geschäftsobjekten, die an der Ausführung von Aktivitäten beteiligt sind, also erzeugt, ausgelesen, manipuliert oder gelöscht werden (siehe Abbildung 8.3(a)). Ereignisse in EPKs wie z. B. „Auftrag angelegt“ und „Auftrag storniert“ in Abbildung 8.3(b) haben vielfach implizit den Charakter einer Zustandsbeschreibung.

Wie wir bereits in Abschnitt 2.3 festgestellt haben, weisen aktuelle Sprachen zur Geschäftsprozessmodellierung einen gemeinsamen Kern von Modellierungskonstrukten auf, mit denen sich betriebliche Aktivitäten beschreiben lassen. Zu diesen Konstrukten gehören neben dem auszuführenden betrieblichen Vorgang die für dessen Ausführung verantwortliche Organisationseinheit sowie die von ihm benutzten, manipulierten oder erzeugten Ressourcen. Die Modellierung einer Aktivität mittels dieser Konstrukte kann ähnlich wie die Signatur einer Operation als (natürlich-)sprachliches Aussageschema verstanden werden, das eine vorgangsorientierte Beschreibung der fachlichen Semantik der Aktivität definiert. Wenn wir die einer Aktivität zugeordnete Organisationseinheit als Subjekt, den auszuführenden Vorgang als syntaktische Verbindung von Prädikat mit direktem Objekt und die genutzten Ressourcen als indirekte Objekte eines Aussageschemas auffassen, so können wir die fachliche Semantik der einzelnen Aktivitäten aus Abbildung 8.3 durch die Aussageschemata „ein Vertriebsmitarbeiter legt einen Auftrag auf Basis eines Angebots an“ und „ein Vertriebsmitarbeiter storniert einen Auftrag“ ausdrücken. Die Ausführung einer Aktivität entspricht dann wiederum der Aktualisierung eines solchen Aussagesche-



(a) Aktivitätsdiagramm (b) Ereignisgesteuerte Prozesskette

Abbildung 8.3: Einfache Geschäftsprozessmodelle zur Auftragsverwaltung

mas in Form einer Aussage. Wie bereits bei den vorgangsorientierten Beschreibungen der Operationen `createOrder()` und `delOrder()` in Abschnitt 8.1.1 tritt auch in den hier betrachteten Geschäftsprozessmodellen die Situation ein, dass der Begriff „Auftrag“ sowohl in intensionaler Bedeutung in der Bezeichnung des Vorgangs als auch in extensionaler Bedeutung zur Benennung einer Ressource genutzt wird. Diese Differenzierung findet sich in unseren Aussageschemata nicht wieder.

Analog zur vorgangsorientierten Spezifikation der fachlichen Semantik von Operationen ist auch bei der (natürlich-)sprachlichen Beschreibung von Aktivitäten ein terminologisches Modell vonnöten, das den für die Formulierung von Geschäftsprozessmodellen bereitstehenden „Wortschatz“ mit hinreichender Präzision definiert. Die gegenüber zustandsorientierten Spezifikationen niedrigere Präzision (natürlich-)sprachlicher Spezifikationen sowie deren maschinelle Verarbeitbarkeit ist auch hier kritisch zu bewerten.

8.1.3 Bewertung der Spezifikationstechniken

Die vorangegangene Betrachtung alternativer Spezifikationstechniken für die fachliche Semantik von Operationen und Aktivitäten hat Vor- und Nachteile beider Ansätze aufgezeigt. So erreicht der zustandsorientierte Ansatz aufgrund seines formalen Charakters eine hohe Präzision, die allerdings mit vergleichsweise großem Spezifikationsaufwand erkauft wird und zu Lasten der Verständlichkeit der resultierenden Spezifikationen geht. Darüber hinaus können zustandsorientierte Spezifikationen die Veröffentlichung von Implementierungsdetails erfordern. Der vorgangsorientierte Ansatz hingegen weist zwar Schwächen hinsichtlich der erreichbaren Präzision und der maschinellen Verarbeitung auf, er unter-

stützt jedoch die Einbeziehung von Fachexperten in alle Phasen des Softwareentwicklungsprozesses.

Da aus Sicht der geschäftsprozessorientierten Komponentensuche neben den bislang betrachteten Eigenschaften auch der semantischen Vergleichbarkeit von Operationen und Aktivitäten eine hohe Bedeutung zukommt, haben wir uns für die einheitliche Repräsentation der fachlichen Semantik durch einen vorgangsorientierten Ansatz entschieden. Diese Entscheidung wird auch dadurch gestützt, dass die in Kapitel 7 definierte formale Semantik von Geschäftsprozessmodellen und Komponentenbeschreibungen bereits einen zustandsorientierten Charakter aufweist. Im Unterschied zu der hier vorgestellten, formal geprägten zustandsorientierten Spezifikation wird die Bedeutung einzelner Zustände dort allerdings allenfalls sehr unpräzise durch die Vergabe von Zustandsnamen spezifiziert.

8.1.4 Übersicht über relevante Forschungsarbeiten

Für die Umsetzung eines vorgangsorientierten Ansatzes erscheint die Betrachtung der Forschungsergebnisse aus den Bereichen der *Ontologien*, des *Wissensmanagements* und der *kontrollierten Sprachen* interessant. Im Folgenden gehen wir überblicksartig auf ausgewählte Arbeiten aus diesen Forschungsgebieten ein.

Auf dem Gebiet der *Ontologien* sind z. B. im *Process Specification Language (PSL) Project* [SGT+00], dem *Enterprise Project* [UKMZ98] und dem *Toronto Virtual Enterprise (TOVE) Project* [FCF93] komplexe *Unternehmensontologien* entwickelt worden, die mittels Basiskonzepten zur Repräsentation von Zuständen, Zeit, Objekten, Aktivitäten, Ressourcen und Effekten der Ausführung dieser Aktivitäten sowie Organisationseinheiten die Modellierung von Organisationsstrukturen, Prozessen und Produkten gestatten. Obgleich die gewählten Konzeptualisierungen als Grundlage eines terminologischen Modells interessant sind, baut unsere Arbeit nicht auf einer derartigen Unternehmensontologie auf, da diese primär auf die Ausführung von Prozessen ausgerichtet sind und temporale Aspekte betonen. Darüber hinaus unterstützen die genannten Projekte nur indirekt die Erstellung von Spezifikationen mit natürlichsprachlichem Charakter.

Das *Open Information Model (OIM)* [MDC99] der inzwischen aufgelösten *Meta Data Coalition* stellt eine umfassende Spezifikation von Metadatentypen für die Beschreibung der in Unternehmen anfallenden Informationen dar. Neben einer umfangreichen Sammlung von Metadatentypen für das Data Warehousing beinhaltet der Vorschlag zur Version 1.1 des OIM auch einige Metadatentypen aus dem Bereich des *Wissensmanagements*. Das OIM definiert hier zum einen Strukturen für den Aufbau eines kontrollierten, d. h. systematisch verwalteten Vokabulars und zum anderen ein einfaches linguistisches Modell für die Ablage von Informationen über Entitäten, deren Beziehungen untereinander sowie Aussagen über diese Beziehungen. Die entsprechenden Metadatentypen erscheinen aus Sicht der vorgangsorientierten Spezifikation der fachlichen Semantik von Operationen und Aktivitäten grundsätzlich interessant. Während Teile des OIM nach der Auflösung der Meta Data Coalition im September 2000 in das *Common Warehouse Model (CWM)* der OMG integriert worden sind, ist der aus unserer Sicht relevante Bereich des Wissensmanagements nicht weitergeführt worden.

Kontrollierte Sprachen entstehen aus natürlichen Sprachen wie z. B. Deutsch oder Englisch, indem die Syntax der zugrunde liegenden Sprache stark vereinfacht und der

Wortschatz auf eine kompakte Menge von Wörtern begrenzt wird, denen eine (oder wenige) eindeutige Bedeutungen zugewiesen werden (*kontrolliertes Vokabular*). Sie werden mit dem Ziel eingesetzt, die Kommunikation zwischen menschlichen und maschinellen Akteuren gleichermaßen zu vereinfachen und präziser zu gestalten. So setzt die Firma *Caterpillar* das von ihr entwickelte *Caterpillar Technical English (CTE)* [KAMN98] für die Erstellung von Dokumentationen ein, die maschinell in verschiedene Zielsprachen übersetzt werden können. Die *European Association of Aerospace Industries (AECMA)* hat mit *AECMA Simplified English* [ATA02] eine kontrollierte Sprache entwickelt, um die Verständlichkeit englischer Dokumentationen im Flugzeugbau zu verbessern. *Attempto Controlled English (ACE)* [FS96b] ist eine maschinell verarbeitbare Spezifikationssprache für die Erstellung von Anforderungsdokumenten, die auf der natürlichen englischen Sprache basiert. ACE kombiniert die Vertrautheit natürlicher Sprachen mit der Präzision formaler Sprachen: ACE-Spezifikationen sind textuelle Repräsentationen formaler Spezifikationen. In die Gruppe der kontrollierten Sprachen sind auch die so genannten Normsprachen einzuordnen, die wir für die vorgangsorientierte Spezifikation fachlicher Semantik nutzen wollen und im Folgenden eingehender betrachten werden.

8.2 Normsprachen

Normsprachen [Ort97, Sch97] können als kontrollierte Sprachen angesehen werden, die ihren Ursprung in der konstruktiven Wissenschaftstheorie haben und durch methodische Rekonstruktion der in einem Anwendungsbereich gesprochenen natürlichen (Fach-)Sprache entwickelt werden. In diesem Abschnitt stellen wir ausgehend von einer allgemeinen Einführung die charakteristischen Aspekte einer Normsprache vertiefend vor.

8.2.1 Einführung

Die *konstruktive Wissenschaftstheorie* [Lor87, Büt95] geht davon aus, dass Fachsprachen nicht losgelöst von natürlichen Sprachen entwickelt werden können, sondern durch methodische *Rekonstruktion* einer natürlichen Sprache zu gewinnen sind. Dabei bezeichnet der Begriff der Rekonstruktion die Klärung und eindeutige Definition der Bedeutung von Wörtern, ihrer Beziehungen untereinander sowie der Regeln für ihren Gebrauch. In diesem Zusammenhang ist auch der Grundstock der Logik zu errichten und z. B. die Bedeutung der Abstraktion zu klären. Eine zentrale Annahme der konstruktiven Wissenschaftstheorie besteht damit darin, dass alles, was mittels einer (formalen) Fachsprache ausgedrückt werden soll, bereits mit den in natürlichen Sprachen enthaltenen Sprachelementen formuliert werden kann. Diese Elemente bedürfen lediglich einer sorgfältigen Rekonstruktion, um so genannte *Sprachdefekte* wie Mehrdeutigkeiten, Ungenauigkeiten oder Widersprüche zu beheben.

Wedekind und Ortner schlagen in [WO80, Wed92] den fachlichen Systementwurf als Anwendungsgebiet des konstruktiven Sprachaufbaus in der Informatik vor. Sie betrachten dabei fachliche Anwendungsbereiche von Informationssystemen als sprachliche Gebilde und verfolgen einen „sprachkritischen“, d. h. auf die Klärung der Struktur und des Inhalts von sprachlichen Aussagen über das jeweilige Fachgebiet ausgerichteten Ansatz zum Software Engineering. Dabei wird ausgehend von einer vorhandenen Benutzer-

fachsprache über eine reglementierte, quasi-natürliche Sprache zunächst ein Fachentwurf erstellt und schließlich eine Implementierung angefertigt. Neben den in der Informatik bei (formalen) Sprachen typischerweise betrachteten Dimensionen *Syntax* (Beziehungen zwischen Wörtern als formaler Aspekt von Aussagen über ein Anwendungsgebiet) und *Semantik* (intensionale oder extensionale Bedeutung von Begriffen und Aussagen als inhaltlicher Aspekt) hebt Ortner die Bedeutung der *Pragmatik* (Geltungssicherung von Begriffen und Aussagen) hervor:

> »*Zu einer ganzheitlichen Applikationssoftware-Entwicklung gehören ihre Planung, orientiert an Geschäftsprozessen, die kontrollierte Einführung der Systeme in die Arbeitsabläufe des Unternehmens und die Sicherstellung ihrer Nutzung im Rahmen der auf begrifflicher (Schema-)Ebene festgelegten Fachterminologie (Fachbegriffe).*« [Ort97]

Diese für die Akzeptanz und Relevanz von Entwicklungsergebnissen wichtige Geltungssicherung von Begriffen und Aussagen soll durch deren systematische Rekonstruktion mit den „Begriffsbenutzern“ in den Anwendungsbereichen sowie die anschließende Implementierung „sachgerechter“ Anwendungssoftware erreicht werden.

In der Systementwicklung kann zwischen formalen und materialen Entwicklungssprachen unterschieden werden. Die Grammatik einer *formalen Sprache* repräsentiert *Themenwörter* (*Fachwörter*) wie „Auftrag“, „Kunde“ oder „planen“ nur durch schematische Variablen, d. h. eine formale Sprache kennt nur so genannte *Partikeln* (*Strukturwörter*) wie z. B. „und“ oder „alle“. Die schematischen Variablen werden erst bei Anwendung der Sprache auf einen Wirklichkeitsausschnitt als Wörter einer Fachsprache interpretiert. *Materiale Sprachen* dagegen reglementieren neben der Verwendung von Partikeln (formaler Teil) auch den Einsatz zulässiger Fachwörter (materialer Teil) bei der Bildung korrekter Aussagen. Eine mit einer materialen Sprache beschriebene fachliche Softwarelösung muss daher nicht nur die formalen, durch die Grammatik vorgegebenen Satzstrukturen einhalten, sondern auch auf dem rekonstruierten und durch das *Lexikon* kontrollierten materialen Wortschatz eines Anwendungsgebiets aufbauen [Ort97].

Der Begriff der *Normsprache*[1] bezeichnet eine materiale Fachsprache eines Unternehmens oder einer Branche, die hinsichtlich ihrer Syntax und Semantik eindeutig festgelegt ist und deren Geltung organisationsweit durchgesetzt wird. Aufbauend auf einer so genannten *Gegenstandseinteilung*, die das dem angestrebten Fachentwurf zugrunde liegende Lösungsparadigma beschreibt, verfügt eine solche Normsprache über eine eingeschränkte Grammatik und einen reduzierten Wortschatz, weist aber dennoch einen natürlichsprachlichen Charakter auf. Im fachlichen Systementwurf dienen Normsprachen als Werkzeuge, mit denen die „Sprachlücke“ zwischen den gewachsenen, natürlichen Sprachen der Anwender und den konstruierten, künstlichen Sprachen der Entwickler geschlossen werden soll. Der Prozess des *normsprachlichen Systementwurfs* kann durch folgende Phasen beschrieben werden (vgl. auch [Ort97, Sch97]):

[1] Lorenzen [Lor87] hat ursprünglich den Begriff der „Orthosprache“ (orthos, griech. richtig) zur Bezeichnung von Fachsprachen eingeführt, die durch Rekonstruktion aus einer Gebrauchssprache hervorgegangen sind. Um nicht den Eindruck einer einzig „richtigen“ Sprache zu erwecken, benutzen Autoren wie Ortner und Schienmann die Bezeichnung „Normsprache“.

1. *Aussagensammlung*: Gewinnung relevanter Aussagen über Sachzusammenhänge in dem betrachteten Anwendungsbereich durch Methoden der Begriffsrekonstruktion wie z. B. Beobachtung, Mitarbeit, Lektüre von Fachliteratur oder Interviews.

2. *Wörterbestimmung*: Rekonstruktion der maßgeblichen Fachbegriffe mit Hilfe einer Normsprache, um Sprachdefekte zu beseitigen.

3. *Aussagennormierung*: Syntaktisch-formale Standardisierung der relevanten Aussagen nach den Regeln einer normsprachlichen Grammatik.

4. *Übergang zu Entwurfssprache*: Übertragung der „bereinigten" Aussagen in eine spezifische Entwurfssprache (z. B. eine Diagrammsprache), um zu einer kürzeren und präziseren Darstellung des Fachentwurfs zu gelangen.

Im Folgenden stellen wir die Aspekte *Gegenstandseinteilung*, *Grammatik* und *Lexikon*, durch die Normsprachen charakterisiert sind, detaillierter vor.

8.2.2 Gegenstandseinteilung

Grundlage jeder Normsprache ist eine *Gegenstandseinteilung*, die als konzeptionelle Einteilung des betrachteten Wirklichkeitsausschnitts ein für die jeweilige Aufgabenstellung geeignetes Lösungsprinzip repräsentiert. Dem Datenbankentwurf nach dem Entity-Relationship-Modell liegt beispielsweise eine Gegenstandseinteilung mit Dingen (Entities) und Beziehungen (Relationships) zugrunde, während beim Entwurf wissensbasierter Systeme Kategorien wie „Fakten", „Regeln" und „Schlussarten" betrachtet werden. Im objektorientierten Paradigma wiederum liegt eine Einteilung in Objekte, Eigenschaften und Geschehnisse (Methoden) vor. Grundsätzlich sind abhängig von kulturellem Hintergrund und verfolgter Absicht verschiedenste Gegenstandseinteilungen denkbar. Abbildung 8.4 zeigt Ortners Vorschlag einer „methodenneutralen", d. h. nicht auf eine bestimmte Softwareentwicklungsmethode zugeschnittenen Gegenstandseinteilung für den Fachentwurf.

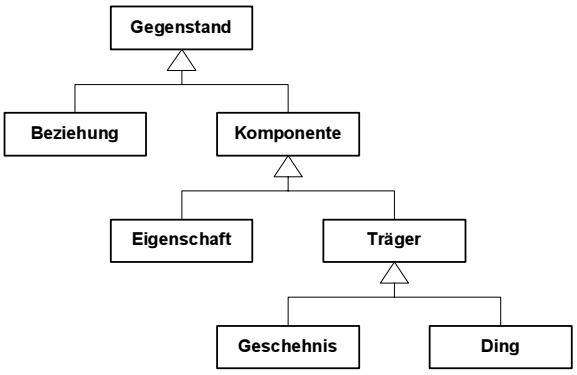


Abbildung 8.4: Methodenneutrale Gegenstandseinteilung nach Ortner [Ort97]

Dabei werden ausgehend vom allgemeinen Begriff des *Gegenstands* zunächst *Komponenten* sowie *Beziehungen* zwischen diesen Komponenten unterschieden. Komponen-

ten können *Eigenschaften* oder *Träger* dieser Eigenschaften darstellen, die wiederum in *Dinge* und *Geschehnisse* eingeteilt werden. SCHIENMANN stellt in [Sch97] eine weitere Differenzierung der Klasse der Geschehnisse vor, die in verkürzter Form in Abbildung 8.5 dargestellt ist. Abhängig davon, ob ein Geschehnis veranlasst ist (z. B. durch eine explizite Aufforderung wie „Leihe das Buch aus!“) oder nicht, lassen sich *Tun* und *Bewegung* unterscheiden. Wird mit einem Tun eine bestimmte Absicht verfolgt, so spricht man von einer *Handlung*, anderenfalls von einem *Verhalten*.

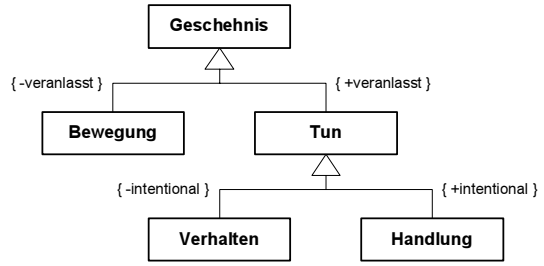


Abbildung 8.5: Differenzierte Einteilung von Geschehnissen nach SCHIENMANN [Sch97]

Zentrale Konstruktionsprinzipien für Gegenstände im Allgemeinen, also sowohl Dinge als auch Geschehnisse, sind durch *Abstraktion* und *Komposition* gegeben. Dabei führt die Abstraktion von individuellen, konkreten Gegenständen zu deren Typen. Eine Handlung ist damit als Aktualisierung eines Handlungstyps zu betrachten, genauso wie ein Ding eine Aktualisierung eines Dingtyps darstellt. Orthogonal zur Abstraktion lassen sich mittels Komposition einerseits Dinge oder Dingtypen ähnlich einer Stückliste zu komplexeren Dingen bzw. Dingtypen und andererseits Handlungen oder Handlungstypen im Sinne der hierarchischen Prozessmodellierung zu komplexeren Handlungen bzw. Handlungstypen zusammensetzen.

8.2.3 Formaler Teil: Grammatik

Eine rationale, d. h. durch methodische Rekonstruktion entwickelte *Grammatik* einer Normsprache besteht aus einer Einteilung der verwendeten *Wortarten* sowie einer Menge so genannter *Satzbaupläne*, die die Bildung von Sätzen reglementieren.

Wortarten Da die Wortarten einer Normsprache der sprachlichen Erfassung eines betrachteten Wirklichkeitsausschnitts dienen, besteht zwischen der Untergliederung der Wortarten und der gewählten Gegenstandseinteilung ein enger Zusammenhang. Abbildung 8.6 zeigt eine Einteilung normsprachlicher Wortarten, die sich an der Gegenstandseinteilung aus Abbildung 8.4 orientiert. Dabei wird zunächst einmal zwischen *Partikeln*, die den Beziehungen der Gegenstandseinteilung zugeordnet werden können, und *Themenwörtern* unterschieden, die den Komponenten der Gegenstandseinteilung entsprechen. Partikeln können weiter differenziert werden in *grammatische Partikeln*, zu denen u. a. die als Hilfsverben dienenden *Kopulae* wie „ist“, „hat“, „kann“ oder „tut“ zählen, und *logische Partikeln* wie Junktoren („und“, „oder“) und Quantoren („für alle“), mit denen sich einfach strukturierte Sätze zu komplexeren Sätzen logisch verknüpfen lassen. Bei den

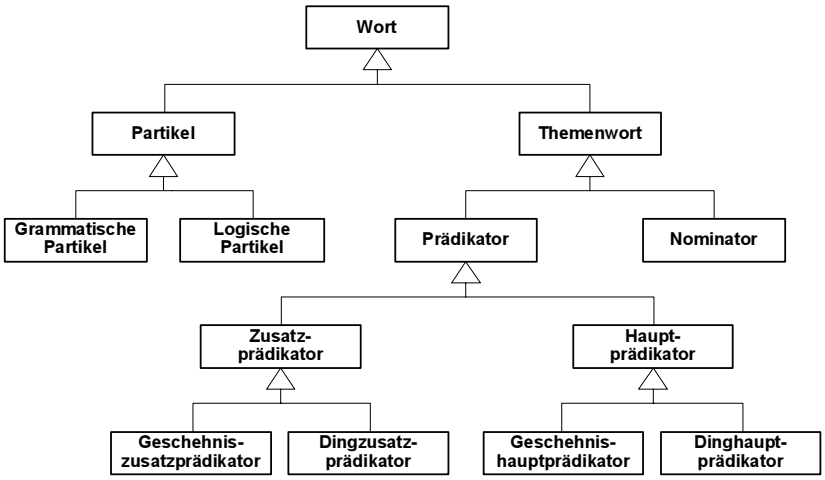


Abbildung 8.6: Normsprachliche Wortarten (vgl. [Sch97])

Themenwörtern unterscheidet man *Nominatoren* zur eindeutigen Benennung einzelner, konkreter Gegenstände durch Eigennamen wie z. B. „Müller“, Leernamen („ich“, „du“) und mittels Demonstratoren gebildete Kennzeichnungen („dieser Auftrag“) von den so genannten *Prädikatoren*, die Typen von Gegenständen oder deren Eigenschaften bezeichnen. Die *Prädikation* als Unterscheidungshandlung verbindet nicht-sprachliches und sprachliches Handeln, indem einem Gegenstand(-styp) eine Eigenschaft zu- oder abgesprochen wird („Dieser Auftrag ist ausgeführt.“). Als sprachliche Hilfsmittel für solche elementaren Prädikationen dienen dabei die oben eingeführten Kopulae. Prädikatoren lassen sich weiter in *Hauptprädikatoren (Eigenprädikatoren)* und *Zusatzprädikatoren (Apprädikatoren)* differenzieren, wobei erstere den Trägern und letztere den Eigenschaften der Gegenstandseinteilung zugeordnet werden. Die spezielleren *Ding-* und *Geschehnishauptprädikatoren* bezeichnen Dinge („Auftrag“) bzw. Geschehnisse („anlegen“), deren Eigenschaften mittels *Ding-* bzw. *Geschehniszusatzprädikatoren* wie „ausgeführt“ oder „erfolgreich“ beschrieben werden können.

Satzbaupläne Im Gegensatz zu der im Allgemeinen nicht eindeutigen Grammatik einer natürlichen Sprache zeichnet sich die rationale Grammatik einer Normsprache dadurch aus, dass ihre Satzbildungsregeln (die so genannten Satzbaupläne) eine Tiefenstruktur definieren, die vergleichbar mit einem Ableitungsbaum eindeutig aus der syntaktischen Struktur eines Satzes abgeleitet werden kann. Angesichts der Vielfalt denkbarer normsprachlicher Satzbaupläne (vgl. z. B. [Sch97]) beschränken wir uns in diesem Abschnitt auf eine knappe Einführung der wesentlichen Konzepte.

Hartmann unterscheidet in [Har90] grundsätzlich zwischen *Imperativsätzen* (Aufforderungen) und *Indikativsätzen* (Aussagen). Auf Snell [Sne52] geht eine Klassifizierung von Indikativsätzen in *Ordnungs-*, *Teilungs-* und *Geschehnisaussagen* zurück, die sich an der in Abbildung 8.4 dargestellten Gegenstandseinteilung in Dinge, Eigenschaften und Geschehnisse orientiert. Dabei ermöglichen Ordnungs- und Teilungsaussagen die Rekonstruktion der statischen Struktur eines Anwendungsbereichs, d. h. die Definition

der relevanten Konzepte eines Anwendungsbereiches sowie deren semantische Beziehungen untereinander. Geschehnisaussagen gestatten die Beschreibung des Verhaltens eines Systems innerhalb dieses Anwendungsbereichs, d. h. die beobachtbaren Interaktionen zwischen Instanzen der Konzepte [Ort97]. Imperativsätze und die verschiedenen vorgestellten Klassen der Indikativsätze unterscheiden sich hinsichtlich der eingesetzten Kopulae. So werden Ordnungsaussagen durch Verwendung der *Seinskopula* „ist“, Teilungsaussagen mit Hilfe der *Teilungskopula* „hat“ und Tataussagen als spezielle Form von Geschehnisaussagen (vgl. Abbildung 8.5) durch die *Tatkopula* „tut“ gebildet, die in einer Normsprache durch die Symbole ϵ, ν und π repräsentiert werden. SCHIENMANN [Sch97] ergänzt u. a. die Gruppe der *Fähigkeitsaussagen*, die durch die *Fähigkeitskopula* „kann“ (gekennzeichnet durch das Symbol γ) charakterisiert sind. Imperativsätze oder Aufforderungen werden mittels des so genannten *Appellators* „!“ formuliert, der als „bitte“ gelesen wird. Tabelle 8.1 zeigt jeweils ein Beispiel für eine Ordnungs-, Teilungs-, Tat- und Fähigkeitsaussage sowie eine Aufforderung in ihrer Repräsentation in natürlicher Sprache, in Normsprache und als Satzbauplan. In der schematischen Darstellung der Satzbaupläne bezeichnet dabei N einen Nominator, Q einen Dinghauptprädikator und P einen Geschehnishauptprädikator.

Satzart / Repräsentation	**Beispiel**
Ordnungsaussage	
natürliche Sprache	„Müller ist ein Mitarbeiter“
Normsprache	`[ Müller | ` ϵ ` | Mitarbeiter ]`
Satzbauplan	`[ N |` ϵ `| Q ]`
Teilungsaussage	
natürliche Sprache	„Müller hat eine Personalnummer“
Normsprache	`[ Müller |` ν `| Personalnummer ]`
Satzbauplan	`[ N |` ν `| Q ]`
Tataussage	
natürliche Sprache	„Müller bearbeitet einen Vorgang“
Normsprache	`[ Müller |` π `| bearbeiten | Vorgang ]`
Satzbauplan	`[ N |` π `| P | Q ]`
Fähigkeitsaussage	
natürliche Sprache	„Müller kann eine Rechnung buchen“
Normsprache	`[ Müller |` γ `| buchen | Rechnung ]`
Satzbauplan	`[ N |` γ `| P | Q ]`
Aufforderung	
natürliche Sprache	„Müller, stornieren Sie bitte einen Auftrag!“
Normsprache	`[ Müller | ! | stornieren | Auftrag ]`
Satzbauplan	`[ N | ! | P | Q ]`

Tabelle 8.1: Beispiele für Aussagen und Aufforderungen

Bei den in Tabelle 8.1 exemplarisch vorgestellten Sätzen handelt es sich um *singuläre* Aussagen bzw. Aufforderungen, durch die ein durch den Nominator N in der Subjektstellung bezeichneter individueller Gegenstand näher beschrieben wird. Für die Entwicklung eines Schemas ist man jedoch an *allgemeinen* Aussagen und Aufforderungen interessiert, bei denen einem Dinghauptprädikator Q in der Subjektstellung eine Eigenschaft zu- oder abgesprochen wird [Ort97]. Als Beispiel für eine solche allgemeine Aussage mag der Satz „Mitarbeiter haben eine Personalnummer“ dienen, der normsprachlich als `[ Mitarbeiter | ν | Personalnummer ]` formuliert und durch den Satzbauplan `[ Q₁ | ν | Q₂ ]` schematisiert werden kann.

Die bislang betrachteten normsprachlichen Sätze weisen eine vergleichsweise einfache Struktur auf, bei der insbesondere keine Themenwörter in der Rolle indirekter Objekte auftreten. Indirekte Objekte werden in Normsprachen durch so genannte *Kasusmorpheme* eingeleitet, die den Fall des jeweiligen direkten Objekts anzeigen. Nach Lorenzen [Lor87] lassen sich zumindest die folgenden drei Fälle unterscheiden:

- *Mittelfall*: Das indirekte Objekt dient als Hilfsmittel zur Durchführung einer Handlung. Der Mittelfall wird durch das Kasusmorphem „mit“ gekennzeichnet, das dem indirekten Objekt vorangestellt wird (`[ Mitarbeiter | π | anlegen | Auftrag | mit-Angebot ]`).

- *Werkfall*: Das indirekte Objekt stellt das Ergebnis der Ausführung einer Herstellungs- oder Transformationshandlung dar. Der Werkfall wird spezifiziert, indem das indirekte Objekt mit dem Morphem „zu“ kombiniert wird (`[ Mitarbeiter | π | überführen | Kundenauftrag | zu-Produktionsauftrag ]`).

- *Gebefall*: Das indirekte Objekt beschreibt das Ziel der Ausführung einer Gebehandlung. Das Kasusmorphem „an“ zeigt in einem normsprachlichen Satz den Gebefall an (`[ Mitarbeiter | π | geben | Angebot | an-Kunde ]`).

Der Satzbauplan für allgemeine Tataussagen mit indirekten Objekten lautet `[ Q₁ | π | P | Q₂ | Q₃^Fall ]`, wobei der verwendete Fall durch Symbole `I`, `II` und `III` für den Mittel-, Werk- und Gebefall gekennzeichnet wird. Sollen mehrere Dinghauptprädikatoren an der Stelle des indirekten Objekts genannt werden, so sind diese durch Semikola zu trennen. Komplexere Aussagen lassen sich u. a. durch die Verwendung von Zusatzprädikatoren zur näheren Beschreibung der benutzten Hauptprädikatoren sowie die Kombination einfacherer Aussagen mittels logischer Partikeln konstruieren (siehe auch [Sch97]).

8.2.4 Materialer Teil: Lexikon

Um dem durch eine Menge von Satzbauplänen gegebenen formalen Rahmen einer Normsprache „Leben einzuhauchen“, ist ein gewisses Vokabular („Material“) erforderlich. Normsprachen greifen zu diesem Zweck auf ein *Lexikon* zurück, in dem ein System methodisch rekonstruierter Themenwörter verwaltet wird. Wir nennen ein Wort, dessen Bedeutung inhaltlich geklärt und definiert worden ist, einen *Begriff* und bezeichnen ein System solcher Begriffe als *Terminologie*. Die Klärung und Festlegung der Bedeutung einzelner Wörter lässt sich mittels der folgenden Verfahren erreichen [Sch97]:

- *Exemplarische Einführung*: Situationsabhängiges Einüben der Benutzung von Prädikatoren durch empraktisches Reden, d. h. durch Verbindung von sprachlichem mit nicht-sprachlichem Handeln.
- *Prädikatorenregeln*: Regelung der Verwendung von Prädikatoren untereinander durch (prädikatenlogische) Regeln.
- *Explizite Definition*: Vollständige und abgeschlossene Festlegung der Bedeutung und Verwendungsweise von Wörtern durch Formulierung logischer Äquivalenzen.

Da die exemplarische Einführung von Themenwörtern nur dann erforderlich ist, wenn eine Einführung durch sachbezogenes, ausschließlich epipraktisches Reden, d. h. durch Reden über nicht-sprachliches Handeln nicht möglich ist, verzichten wir hier auf ihre detaillierte Vorstellung. Stattdessen konzentrieren wir uns auf die Klärung und Festlegung der Bedeutung von Wörtern mittels Prädikatorenregeln und expliziten Definitionen.

Prädikatorenregeln reglementieren den Gebrauch von Prädikatoren durch die Definition von semantischen Beziehungen zwischen Prädikatoren bzw. den Gegenständen, denen diese Prädikatoren zu- oder abgesprochen werden. Dabei wird durch Regeln, deren Aufbau sich weitestgehend an der Prädikatenlogik orientiert, einem Gegenstand in Abhängigkeit von der Verwendbarkeit eines gegebenen Prädikators ein anderer Prädikator zugesprochen (affirmativer Fall) oder abgesprochen (negativer Fall) [Sch97]. Als Beispiel für eine Prädikatorenregel soll die folgende *Subordinationsregel* dienen, die besagt, dass jeder Gegenstand, der einen Kundenauftrag darstellt, auch als Auftrag bezeichnet werden kann.

$$x \in Kundenauftrag \quad \Rightarrow \quad x \in Auftrag$$

Mit derartigen Subordinationsregeln der Form $x \in P_1 \Rightarrow x \in P_2$ wird eine für die Abstraktion charakteristische Inklusionsbeziehung zwischen Begriffen definiert, mit der z. B. vom Dinghauptprädikator „Mitarbeiter“ zum Prädikator „Person“ oder vom Geschehnishauptprädikator „übermitteln“ zum Prädikator „kommunizieren“ abstrahiert wird. Weitere, häufig verwendete Prädikatorenregeln dienen der Formulierung von *Synonymie* ($x \in P_1 \Leftrightarrow x \in P_2$) oder *Kontrarität* ($x \in P_1 \Rightarrow x \in' P_2$ und $x \in P_2 \Rightarrow x \in' P_1$), wobei die Bedeutung des Operators $\in'$ natürlichsprachlich durch „ist kein“ oder „ist nicht“ gegeben ist.

Explizite Definitionen legen die Bedeutung und Verwendungsweise von Prädikatoren fest, indem eine logische Äquivalenz zwischen dem zu definierenden Prädikator (dem *Definiendum*) und seiner Definition (dem *Definiens*) spezifiziert wird. Das grundlegende Schema expliziter Definitionen lautet $x \in P :\Leftrightarrow A(x)$, wobei alle in der Aussage $A(x)$ genutzten Begriffe bereits definiert sein müssen. Das Definiendum wird damit gleichsam als abkürzende Schreibweise für das Definiens eingeführt. Als Beispiel für eine explizite Definition soll hier die Einführung des Begriffs „Handlung“ als beabsichtigtes Tun dienen (vgl. auch Abbildung 8.5):

$$x \in Handlung \quad :\Leftrightarrow \quad x \in Tun \ \wedge \ x \in Absicht$$

Als eine prinzipielle Möglichkeit der Geltungssicherung eines sprachlich nicht vorausgesetzten definitorischen Ankers für die Erstellung expliziter Definitionen kann die

exemplarische Einführung betrachtet werden. SCHIENMANN betont jedoch, dass normsprachliche Definitionen – ähnlich der in Programmiersprachen üblichen Konstruktion komplexer Datentypen auf der Grundlage vordefinierter Basisdatentypen – von Basisprädikatoren wie „Person“, „Buch“ oder „ausleihen“ und „zurückgeben“ ausgehen können, ohne dass deren Gebrauch im Fachentwurf empraktisch eingeübt werden muss [Sch97].

8.3 Normsprachliche Repräsentation der fachlichen Semantik

Ziel dieses Abschnitts ist die Entwicklung einer Normsprache, auf deren Grundlage sich die fachliche Semantik der in Geschäftsprozessmodellen spezifizierten Aktivitäten einerseits sowie der von Komponenten angebotenen Operationen andererseits in einer vereinheitlichten Form repräsentieren lässt. Eine solche Normsprache soll wie in Abbildung 8.7 dargestellt die gemeinsame fachliche Basis der im Rahmen der geschäftsprozessorientierten Komponentensuche eingesetzten Sprachen der Verwender und Hersteller bilden und schließlich bei der semantischen Komponentensuche als fachlicher Mittler dienen.

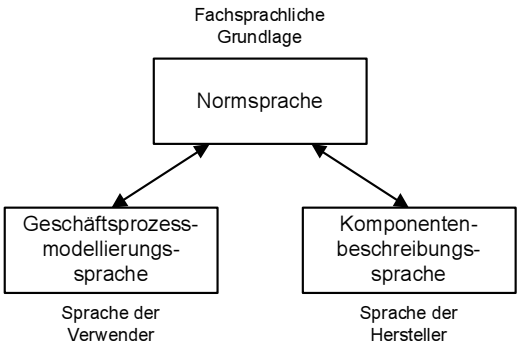


Abbildung 8.7: Normsprachliche Basis der Sprachen von Verwendern und Herstellern

Wir rekonstruieren im Folgenden zunächst die Grundlagen einer geeigneten Normsprache aus den Eigenschaften der bei der geschäftsprozessorientierten Komponentensuche eingesetzten Sprachen. Aufbauend auf den Ergebnissen dieser Rekonstruktion führen wir anschließend ein terminologisches Modell zur rechnergestützten Verwaltung entsprechender Normsprachen ein.

8.3.1 Rekonstruktion der fachlichen Semantik von Geschäftsprozessmodellen und Komponentenbeschreibungen

Einen auch aus Sicht unserer Arbeit interessanten Vorschlag für die normsprachliche Entwicklung objektorientierter Spezifikationen macht SCHIENMANN mit dem Rahmenwerk *TAOS (Terminologiebasierter Ansatz für die objektorientierte Spezifikation)* [Sch97]. In dem in Abbildung 8.8 dargestellten Spezifikationsrahmen von TAOS werden die Aspekte *Statik*, *Funktionalität* und *Dynamik* objektorientierter Spezifikationen betrachtet, wobei

jeweils zwischen einer *internen* und einer *externen* Objektsicht unterschieden wird. Orthogonal zu diesen Kategorien ist die Gruppe der Einschränkungen angesiedelt, in der zusätzliche Invarianten und Regeln für den objektorientierten Fachentwurf eingeordnet werden.

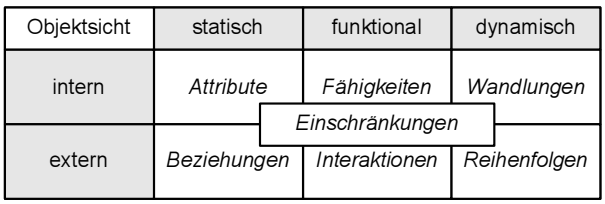
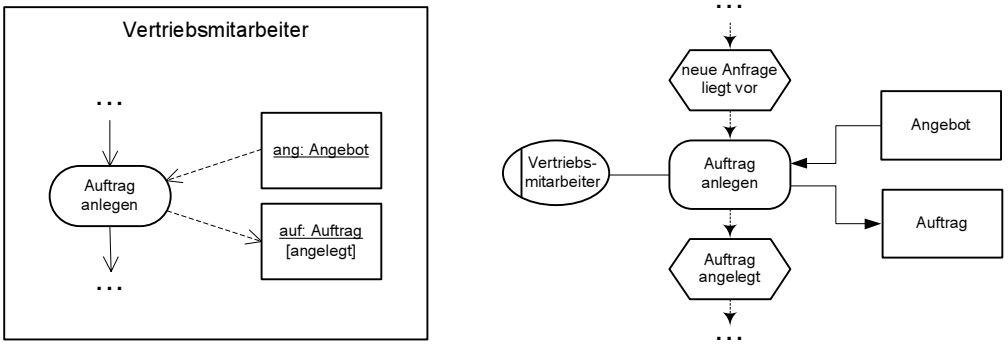


Abbildung 8.8: Rahmen für die objektorientierte Spezifikation nach TAOS (nach [Sch97])

Schienmann rekonstruiert mit TAOS für jede dieser sieben Kategorien objektorientierter Spezifikationen geeignete Konstrukte einer Spezifikationssprache TAOS-S, eine entsprechende Diagrammsprache TAOS-D sowie eine normsprachliche Repräsentation. Im Folgenden orientieren wir uns an der normsprachlichen Rekonstruktion der aus Sicht der geschäftsprozessorientierten Komponentensuche relevanten Kategorien.

Rekonstruktion von Geschäftsprozessmodellierungssprachen Für die normsprachliche Rekonstruktion von Geschäftsprozessmodellierungssprachen im Rahmen unserer Arbeit ist primär die Kategorie der *Reihenfolgen* von Interesse. Reihenfolgen spezifizieren in TAOS die einzelnen Aktivitäten eines Prozesses sowie die kausalen Abhängigkeiten zwischen diesen Aktivitäten. Für die Definition der kausalen Abhängigkeiten als Präzedenzrelation stehen dabei ähnliche Operatoren zur Verfügung, wie sie lineare Prozessmodelle anbieten (vgl. Abschnitt 2.2.3). Da wir in unserem Ansatz die Präzedenzrelation eines Geschäftsprozessmodells bereits durch dessen formale Semantik festhalten (vgl. Abschnitt 7.4), sind wir im Folgenden lediglich an der Rekonstruktion einzelner Aktivitäten interessiert.

Die Spezifikation einer Aktivität in einem Geschäftsprozessmodell kann allgemein als Aufforderung an eine betriebliche Organisationseinheit verstanden werden, eine bestimmte Handlung auszuführen. Wie in Abschnitt 8.2.3 dargelegt können Aufforderungen normsprachlich mit Hilfe des Appellators „!“ rekonstruiert werden. Obgleich mit den verschiedenen in Abschnitt 2.2 vorgestellten Sprachen zur Geschäftsprozessmodellierung unterschiedliche Zielsetzungen verfolgt und damit unterschiedliche Schwerpunkte gesetzt werden, weisen diese Sprachen einen gemeinsamen Kern von Modellierungskonstrukten zur Spezifikation betrieblicher Aktivitäten auf. Wie bereits in Abschnitt 8.1.2 erwähnt lässt sich daher die einer Aktivität zugeordnete Organisationseinheit als Subjekt, der auszuführende Vorgang als syntaktische Verbindung von Prädikat mit direktem Objekt und die Ressourcen als indirekte Objekte eines normsprachlichen Aufforderungsschemas auffassen. Abbildung 8.9 stellt die Modellierung einer betrieblichen Aktivität in einem Aktivitätsdiagramm ihrer Modellierung in einer EPK gegenüber. Die fachliche Semantik dieser Aktivität kann normsprachlich durch die Aufforderung



(a) Aktivität eines Aktivitätsdiagramms (b) Funktion einer EPK

Abbildung 8.9: Repräsentation von Aktivitäten in Aktivitätsdiagrammen und EPKs

`[ Vertriebsmitarbeiter | ! | anlegen | Auftrag | mit-Angebot ]`

wiedergegeben werden. Aus Gründen, die wir bereits in Abschnitt 8.1.2 erläutert haben, haben wir dabei auf die zweite, extensional orientierte Repräsentation des direkten Objekts der Aufforderung als indirektes Objekt (`zu-Auftrag`) verzichtet. Ein allgemeiner Satzbauplan für die normsprachliche Formulierung von betrieblichen Aktivitäten ist damit durch das Aufforderungsschema

$$[\ Q_1 \mid ! \mid P \mid Q_2 \mid Q_{3_1}^{\mathtt{Fall}_1}; Q_{3_2}^{\mathtt{Fall}_2}; \ldots\ ] \tag{8.1}$$

gegeben. Erst die Aktualisierung einer solchen Aufforderung im Rahmen der Prozessausführung ist als Tataussage zu verstehen, deren Struktur durch den Satzbauplan

$$[\ Q_1 \mid \pi \mid P \mid Q_2 \mid Q_{3_1}^{\mathtt{Fall}_1}; Q_{3_2}^{\mathtt{Fall}_2}; \ldots\ ] \tag{8.2}$$

festgelegt ist.

Rekonstruktion von Komponentenbeschreibungssprachen Bei der normsprachlichen Rekonstruktion von Komponentenbeschreibungssprachen beschränken wir uns auf die Betrachtung der in der direkten bzw. den indirekten Schnittstellen einer Komponente angebotenen Operationen. Während die in einer CDL-Komponentenbeschreibung darüber hinaus enthaltenen Verhaltensbeschreibungen bereits durch die formale Semantik einer Komponente oder Klasse repräsentiert werden (vgl. Abschnitt 7.4), sind weitere Aspekte wie z. B. Kompositionsbeziehungen zwischen Komponenten aus Sicht der geschäftsprozessorientierten Komponentensuche nicht relevant.

Schienmann rekonstruiert die fachliche Semantik von Operationen in den Kategorien *Fähigkeiten* und *Wandlungen*, indem er dem Prinzip des Design by Contract folgend neben der normsprachlichen Beschreibung der Operationen auch normsprachliche Aussagen über Vor- und Nachbedingungen der Ausführung dieser Operationen formuliert. Damit wird in TAOS eine Kombination des zustandsorientierten und des vorgangsorientierten Ansatzes zur Spezifikation fachlicher Semantik eingesetzt. Da wir eine zustandsorientierte Sicht auf das Verhalten von Komponenten und Klassen bereits durch die Repräsentation

ihrer formalen Semantik als Transitionssysteme erreichen, verzichten wir im Folgenden auf die normsprachliche Rekonstruktion von Zusicherungen zur Beschreibung der Anwendbarkeit und Wirkung von Operationen.

Die Spezifikation einer Operation in einer Schnittstelle kann allgemein als Fähigkeit der Komponente bzw. Klasse interpretiert werden, einen bestimmten Dienst zu erbringen. In einer Normsprache lassen sich entsprechende Fähigkeitsaussagen mittels der Fähigkeitskopula γ rekonstruieren (vgl. auch Abschnitt 8.2.3) und durch den folgenden Satzbauplan formulieren:

$$[\ \mathtt{Q_1}\ |\ \gamma\ |\ \mathtt{P}\ |\ \mathtt{Q_2}\ |\ \mathtt{Q}_{3_1}^{\mathtt{Fall}_1};\mathtt{Q}_{3_2}^{\mathtt{Fall}_2};\ldots\ ] \tag{8.3}$$

Die Nutzung einer solchen Fähigkeit durch die tatsächliche Ausführung der Operation kann wiederum als Aktualisierung von Satzbauplan 8.2 angesehen werden.

Schienmann stellt beim normsprachlichen Fachentwurf nach TAOS einen engen Zusammenhang zwischen den Fähigkeitsaussagen, die im Rahmen der Aussagensammlung erhoben werden, und den Operationen eines im Fachentwurf zu konstruierenden Typs fest. Demnach entspricht das Subjekt $\mathtt{Q_1}$ einer Fähigkeitsaussage dem Typ, dem die Fähigkeit zugeschrieben wird. Direkte Objekte $\mathtt{Q_2}$ einer Fähigkeitsaussage sind Objekte, die von der Ausführung der Fähigkeit unmittelbar betroffen sind, indem sie erzeugt, gelöscht oder manipuliert werden. Dazu kann auch das Objekt gehören, das die Fähigkeit als Dienst anbietet. Indirekte Objekte $\mathtt{Q}_{3_i}^{\mathtt{Fall}_i}$ sind durch die Ausführung der Fähigkeit nur mittelbar betroffen, d. h. sie werden lediglich ausgelesen, aber nicht verändert.

Im Kontext der Ableitung normsprachlicher Fähigkeitsaussagen aus vorliegenden Komponentenbeschreibungen gestaltet sich die Zuordnung von Elementen der Signatur einer Operation zu Positionen im Satzbauplan schwieriger. Im Gegensatz zum normsprachlichen Fachentwurf nach TAOS liegt der Entwicklung von Komponenten im Allgemeinen keine kanonische Vorgehensweise zugrunde, so dass die normsprachlichen Aussagen nicht einfach aufgrund der zuvor geschilderten Zusammenhänge aus dem vorliegenden Schema (den Komponentenbeschreibungen) gewonnen werden können. Vielmehr existieren für die Belegung des Satzbauplans mit Elementen der Signatur einer Operation verschiedene Alternativen, zwischen denen abhängig vom Einzelfall eine geeignete Auswahl zu treffen ist. Da solche Auswahlentscheidungen jedoch ein tieferes (fachliches) Verständnis des durch eine Komponente oder Klasse implementierten Schemas voraussetzen, das sich nicht algorithmisch aus der Komponentenbeschreibung ableiten lässt, kann diese Auswahl nicht schematisch vorgegeben werden. Tabelle 8.2 fasst Alternativen für die Zuordnung von Signaturbestandteilen zu Subjekt, Prädikat, direktem Objekt sowie indirekten Objekten von Satzbauplan 8.3 zusammen und gibt jeweils ein erläuterndes Beispiel.

Bezüglich der Belegung des Subjekts besteht eine Wahlmöglichkeit, die unabhängig von der betrachteten Operation ist und aus diesem Grund gesondert diskutiert werden soll. Beiden Alternativen liegt die Idee zugrunde, dem Subjekt den mit der Ausführung einer Operation assoziierten Akteur zuzuordnen. Dabei lassen sich hinsichtlich der Rolle des Akteurs grundsätzlich zwei Kandidaten ausmachen. Zum einen könnte der durch einen Aufruf angesprochene Besitzer der Operation (*Callee*), also eine Komponente oder ein Geschäftsobjekt einer Klasse als Akteur betrachtet werden, da dieser die Operation tatsächlich ausführt. Zum anderen könnte auch der Aufrufer einer Operation (*Caller*), in dessen Auftrag die Operation ausgeführt wird, als Akteur gelten. Da der konkrete Aufrufer zum

Satzstellung / Alternative	**Beispiel**
Subjekt	
Komponente/Klasse	`SalesComponent::createInvoice(out inv: Invoice)` → `[ SalesComponent | γ | anlegen | Rechnung ]`
Benutzerrolle	Rolle „Vertrieb“ ist zur Ausführung der Operation `SalesComponent::createInvoice()` berechtigt → `[ Vertrieb | γ | anlegen | Rechnung ]`
Prädikat	
Bezeichner der Operation	`Order::add(in pos: OrderPosition)` → `[ ... | γ | hinzufügen | Auftragsposition ]`
Teil des Bezeichners der Operation	`Order::addOrderPosition(in pos: OrderPosition)` → `[ ... | γ | hinzufügen | Auftragsposition ]`
Direktes Objekt	
Klasse/Objekttyp	`Order::plan()` → `[ ... | γ | planen | Auftrag ]`
Eigenschaft von Klasse/Objekttyp	`Order::calculateDueDate()` → `[ ... | γ | berechnen | Fälligkeit ]`
Parameter	`Order::add(in pos: OrderPosition)` → `[ ... | γ | hinzufügen | Auftragsposition | zu-Auftrag ]`
Teil des Bezeichners der Operation	`OrderManagement::checkOrder(out res: boolean)` → `[ ... | γ | prüfen | Auftrag | zu-Ergebnis ]`
Indirektes Objekt	
Parameter	`OrderManagement::createOrder (in offer: Offer, out order: Order)` → `[ ... | γ | anlegen | Auftrag | mit-Angebot ]`
Klasse/Objekttyp	`Order::add(in pos: OrderPosition)` → `[ ... | γ | hinzufügen | Auftragsposition | zu-Auftrag ]`

Tabelle 8.2: Alternativen für die Zuordnung von Signaturelementen zu Satzbauplan

Zeitpunkt der Spezifikation einer Komponente nicht bekannt ist, ließe sich an seiner Stelle die Rolle der zur Ausführung der Operation berechtigten Benutzer als (abstrakter) Akteur betrachten (vgl. hierzu auch [JT02]). In diesem Zusammenhang ist allerdings zu beachten, dass die Informationen über das Berechtigungskonzept einer Komponente nicht in den Signaturen der Operationen abgelegt sind. Komponententechnologien wie EJB und CCM gestatten in ihren Komponentenbeschreibungen jedoch die Deklaration von Rollen, die Benutzern einer Komponente bei deren Einsatz in einem Anwendungssystem zugewiesen werden können, und die Zuordnung dieser Rollen zu den Operationen einer Komponente im Sinne eines Berechtigungskonzepts. Für folgende Beispiele entscheiden wir uns für die Verwendung des zuletzt genannten Ansatzes, d. h. wir ordnen dem Subjekt die Rolle der zur Ausführung der Operation berechtigten Benutzer zu.

8.3.2 Terminologisches Modell der Normsprache

Der praktische Einsatz einer Normsprache erfordert den Aufbau eines Modells zur Repräsentation von Terminologien, die das Vokabular für die Bildung von Sätzen bereitstellen. Während für Ortner eine solche Terminologie »[...] nichts anderes als die fachliche Theorie des betreffenden Anwendungsgebiets [...]« [Ort97] ist, betont Buitelaar in [Bui01], dass eine Terminologie nur eine von mehreren möglichen sprachlichen Repräsentationen einer *Ontologie* ist. Der Begriff der Ontologie [CJB99, MSS01] stammt ursprünglich aus der Philosophie, wo er die *Lehre vom Sein* bezeichnet, und wird in der Informatik uneinheitlich unter zwei verschiedenen Bedeutungen benutzt. Zum einen wird er verwendet, um wie auch bei Buitelaar ein Vokabular für die Repräsentation von Wissen sowie die dieser Repräsentation zugrunde liegende Konzeptualisierung zu bezeichnen, und zum anderen bezieht er sich auf Wissen über einen Anwendungsbereich selbst. Unser *terminologisches Modell* berücksichtigt beide Bedeutungen:

- Auf der *Konzeptualisierungsebene* spezifizieren wir ein allgemeines, domänenunabhängiges Vokabular für die rechnergestützte Repräsentation normsprachlicher Aussagen.

- Die *Wissensebene* definiert aufbauend auf der Konzeptualisierungsebene Begriffe, mit denen Aussagen über Anwendungsbereiche formuliert werden können, sowie deren Beziehungen untereinander.

Konzeptualisierungsebene Hinsichtlich der Betrachtung einer Ontologie als »explizite Spezifikation einer Konzeptualisierung« (übersetzt nach [Gru93]) herrschen unterschiedliche Auffassungen über den Charakter einer solchen Spezifikation vor. Während einige Autoren eine Konzeptualisierung extensional als Menge von Beziehungen betrachten, mit denen ein bestimmter Zustand des betrachteten Weltausschnitts beschrieben wird (vgl. [GN87]), sehen andere in einer Konzeptualisierung eine intensionale Struktur, mit der sich alle möglichen Zustände des Weltausschnitts erfassen lassen (vgl. [GG95]). Wir teilen letztere Auffassung und stellen auf der Konzeptualisierungsebene ein Vokabular bereit, mit dem sich die in einem betrachteten Anwendungsbereich relevanten Begriffe, deren Beziehungen untereinander sowie die für die Spezifikation von Geschäftsprozessen und Komponenten erforderlichen normsprachlichen Sätze beschreiben lassen.

Grundlage der Wahl der in unserem terminologischen Modell berücksichtigten Wortarten sind die Satzbaupläne 8.1–8.3. Abbildung 8.10 stellt die von uns gewählten Wortarten sowie deren Beziehungen untereinander in Form eines Klassendiagramms graphisch dar. Unsere Wahl unterscheidet sich von der in Abbildung 8.6 vorgestellten Hierarchie von Wortarten insbesondere dadurch, dass wir vereinfachend auf die Betrachtung von Nominatoren und Zusatzprädikatoren verzichten. *Aktivitäten* entsprechen der Kategorie der Geschehnishauptprädikatoren, während *Konzepte* als Dinghauptprädikatoren Gegenstände beschreiben, die von der Ausführung einer Aktivität betroffen sind. *Organisationseinheiten* stellen eine spezielle Form von Konzepten dar, die nicht nur Gegenstand der Ausführung einer Aktivität sein können, sondern als Handlungsträger auch selbst eine Aktivität ausführen dürfen (z. B. „Vertrieb“, „Buchhalter“). Bei den (grammatischen)

Partikeln beschränken wir uns auf die *Kopulae* und ordnen der Kategorie der Partikeln die (Kasus-)*Morpheme* zu.

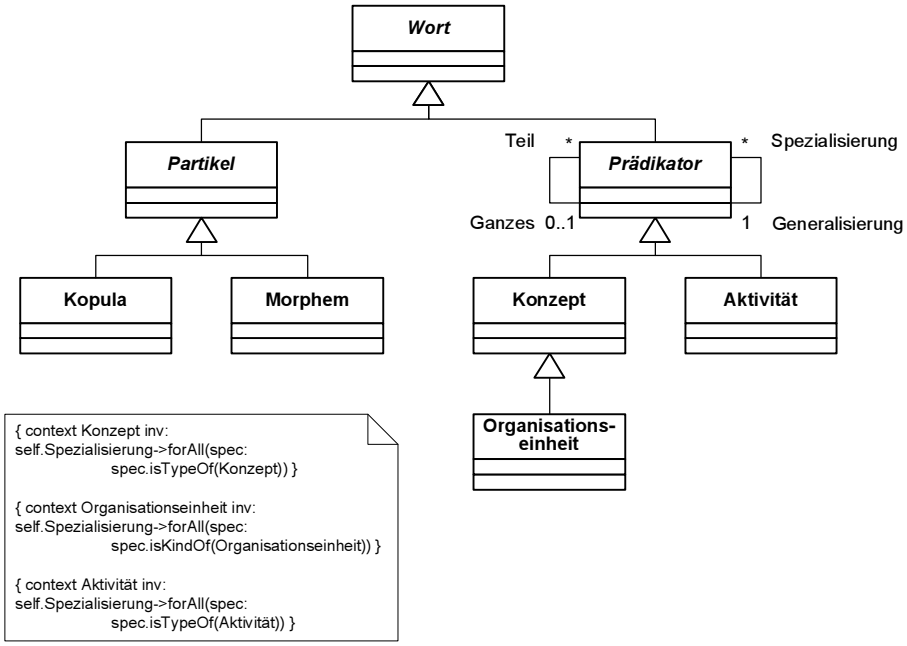


Abbildung 8.10: Wortarten im terminologischen Modell

Zwecks Rekonstruktion der statischen Struktur von Anwendungsbereichen können Prädikatoren in unserem terminologischen Modell zweierlei Beziehungen eingehen, die den normsprachlichen Ordnungs- und Teilungsaussagen entsprechen. So lassen sich Prädikatoren in Generalisierungs-/Spezialisierungsbeziehungen anordnen. Dabei wird allerdings gefordert, dass eine Instanz einer Unterklasse der abstrakten Klasse der Prädikatoren nur durch Instanzen spezialisiert werden kann, die derselben Klasse oder einer Unterklasse angehören. Diese Restriktion möglicher Spezialisierungsstrukturen, die durch die OCL-Ausdrücke in Abbildung 8.10 formalisiert wird, hat beispielsweise zur Folge, dass eine Organisationseinheit nicht durch ein Konzept spezialisiert werden darf. Neben Generalisierungs-/Spezialisierungsbeziehungen berücksichtigen wir in unserem terminologischen Modell auch Ganzes/Teile-Beziehungen zwischen Prädikatoren. Während Generalisierungs-/Spezialisierungsbeziehungen jedoch aufgrund des von uns verfolgten Subtyping-Ansatzes von zentraler Bedeutung für den Vergleich fachlicher Semantik sind, fördern Ganzes/Teile-Beziehungen lediglich die Strukturierung einer Terminologie. Auf eine ähnlich differenzierte Betrachtung von Ganzes/Teile- bzw. Kompositionsbeziehungen, wie sie Schienmann in [Sch97] vorstellt, verzichten wir. Weitergehende Beziehungstypen zwischen Prädikatoren wie z. B. Homonymie oder Synonymie werden in unserem terminologischen Modell ebenfalls nicht betrachtet.

Für die Repräsentation von Aufforderungen, Tat- und Fähigkeitsaussagen gemäß der Satzbaupläne 8.1–8.3 ergänzen wir das terminologische Modell wie in Abbildung 8.11 dargestellt um das Konstrukt der *Sätze*. Ein Satz besteht aus einer Organisationseinheit

in der Subjektstellung, einer Aktivität als Prädikat und Konzepten zur Repräsentation des direkten bzw. der indirekten Objekte, wobei letztere mit Morphemen assoziiert sind. Die Differenzierung zwischen Aufforderungen, Tat- und Fähigkeitsaussagen spiegelt sich in unserem terminologischen Modell durch die Wahl der Kopula wider.

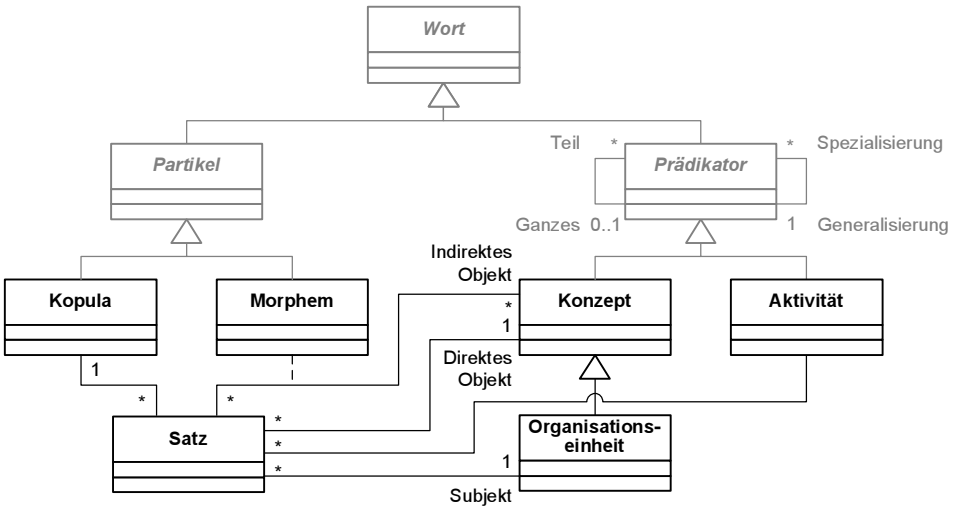


Abbildung 8.11: Satzbauplan im terminologischen Modell

Wissensebene Aufgabe der Wissensebene des terminologischen Modells ist es, die zur sprachlichen Repräsentation eines betrachteten Anwendungsgebietes relevanten Begriffe zu definieren. Aufbauend auf den Generalisierungs-/Spezialisierungsbeziehungen der Konzeptualisierungsebene werden auf der Wissensebene daher neben Kopulae und Morphemen Taxonomien von Konzepten und Aktivitäten spezifiziert. Diese Taxonomien definieren beispielsweise, dass ein „Auftrag“ ein „Geschäftsvorgang“ ist und dass „Produktionsauftrag“ und „Kundenauftrag“ Spezialisierungen ($\leq_K$) des Konzepts „Auftrag“ darstellen. Dabei kennzeichnet das Basiskonzept „etwas“ die Wurzel der Konzepttaxonomie. Auf analoge Weise werden z. B. die Aktivitäten „anlegen“, „ändern“ und „löschen“ als Spezialisierungen ($\leq_A$) der Aktivität „modifizieren“ eingeführt. Der Anker der Aktivitätstaxonomie ist durch die Basisaktivität „tun“ gegeben. Neben der Einordnung in eine Taxonomie ist für jeden Prädikator eine natürlichsprachliche Erklärung anzulegen. Aus normsprachlicher Sicht werden für die Festlegung der Bedeutung von Begriffen folglich die Verfahren der Prädikatorenregeln (Subordination) und der expliziten Definition eingesetzt (vgl. Abschnitt 8.2.4). Mit diesen Kenntnissen über den Gegenstand der Wissensebene lässt sich der Begriff der Terminologie wie folgt formal definieren:

Definition 8.1 (Terminologie) *Eine* Terminologie *ist eine Struktur* $T = (T_K, T_A, \leq_K, \leq_A)$, *wobei*

- T_K *eine endliche Menge von Konzepten,*
- T_A *eine endliche Menge von Aktivitäten,*

- $\leq_K$ *eine Spezialisierungsrelation über* T_K *und*
- $\leq_A$ *eine Spezialisierungsrelation über* T_A *ist.*

Wir schreiben $s \in T$*, um auszudrücken, dass ein normsprachlicher Satz* s *über den Begriffen der Terminologie* T *gebildet wurde.*

Die auf der Wissensebene definierten Begriffe lassen sich für die Bildung normsprachlicher Sätze gemäß der Satzbaupläne 8.1–8.3 nutzen. Als problematisch ist dabei einzuschätzen, dass die Konstruktion von Sätzen lediglich durch syntaktische Anforderungen sowie die verfügbaren Begriffe eingeschränkt wird. Da keine Aussagen über die inhaltliche „Kompatibilität" von Begriffen getroffen werden, lassen sich auch Sätze wie `[ Kunde |`π`| anlegen | Angestellter ]` erzeugen, die aus formaler Sicht zwar durchaus korrekt sein mögen, aber inhaltlich keinen Sinn ergeben. Ortner [Ort97] weist in diesem Zusammenhang auf das Verfahren der *lexikalischen Kategorisierung* hin, bei dem die Anwendbarkeit von Satzbauplänen auf der Grundlage einer kategorialen Einordnung von Wörtern (z. B. Lebewesennominator, Sachprädikator) kontrolliert werden kann. Begrenzend wirkt bei diesem Verfahren, dass lediglich Aussagen über die Kombinierbarkeit lexikalischer Kategorien, nicht aber einzelner Prädikatoren getroffen werden können. Einen Schritt weiter geht ein Ansatz, den wir in [Tes02] beschrieben haben. Dabei werden semantische Restriktionen der Kombination von Prädikatoren mittels einer Menge so genannter *Kernsätze* spezifiziert, die Beispiele für die inhaltlich „korrekte" Kombination von Prädikatoren geben. Über eine spezielle Regel lassen sich unter Berücksichtigung der Konzept- und Aktivitätstaxonomien neue Sätze aus diesen Kernsätzen ableiten, deren Gültigkeit im Hinblick auf die vorgegebenen Kernsätze sichergestellt ist. Da dieser Ansatz für die geschäftsprozessorientierte Komponentensuche nicht von zentraler Bedeutung ist, verzichten wir auf seine detaillierte Einführung.

8.4 Fachliche Semantik von Geschäftsprozessmodellen und Komponentenbeschreibungen

Die fachliche Semantik eines Geschäftsprozessmodells bzw. einer Komponentenbeschreibung ergibt sich aus der fachlichen Semantik der einzelnen geforderten Aktivitäten bzw. angebotenen Operationen. Ziel dieses Abschnitts ist die Definition dieser fachlichen Semantik von Aktivitäten und Operationen auf der Grundlage normsprachlicher Sätze sowie die Herstellung der für die geschäftsprozessorientierte Komponentensuche erforderlichen semantischen Vergleichbarkeit von Aktivitäten und Operationen.

Eine Voraussetzung für die nachfolgende Definition und den Vergleich fachlicher Semantik ist die Existenz einer Halbordnung über normsprachlichen Sätzen. Da wir im Rahmen der geschäftsprozessorientierten Komponentensuche einen Subtyping-Ansatz verfolgen, liegt es nahe, diese Halbordnung als Spezialisierungsrelation anzulegen. Die grundlegende Idee einer solchen Spezialisierungsrelation zwischen normsprachlichen Sätzen ist zu verlangen, dass alle Satzbestandteile (Subjekt, Prädikat sowie direkte und indirekte Objekte) des generelleren Satzes gemäß der Konzept- und Aktivitätstaxonomien durch entsprechende Satzbestandteile des spezielleren Satzes spezialisiert werden. Im Rahmen

der geschäftsprozessorientierten Komponentensuche schränken wir die Spezialisierungsrelation jedoch auf die Betrachtung von Subjekt, Prädikat und direktem Objekt ein, da die Berücksichtigung von indirekten Objekten aus unserer Sicht unangemessen hohe Anforderungen an die semantische Übereinstimmung zweier Sätze stellt und damit die Suchergebnisse zu stark einschränkt. Wir verzichten darüber hinaus auf die Betrachtung der in den zu vergleichenden Sätzen verwendeten Kopulae, da sowohl die Erfüllung einer Aufforderung (Satzbauplan 8.1) als auch die Ausführung einer Operation (Satzbauplan 8.3) als Tataussage (Satzbauplan 8.2) mit der (Tat-)Kopula π zu verstehen ist. Die Spezialisierungsrelation über normsprachlichen Sätzen ist demnach wie folgt definiert:

Definition 8.2 (Spezialisierungsrelation über normsprachlichen Sätzen) *Seien $T = (T_K, T_A, \leq_K, \leq_A)$ eine Terminologie und $s_i = (S_{s_i}, P_{s_i}, O_{s_i}) \in T$, $i \in \{1, 2\}$ Sätze, wobei $S_{s_i} \in T_K$ das Subjekt, $P_{s_i} \in T_A$ das Prädikat und $O_{s_i} \in T_K$ das direkte Objekt des Satzes s_i repräsentieren. Dann gilt:*

$$s_1 \text{ ist spezieller als } s_2 \text{ bzgl. } T \quad (s_1 \leq_T s_2) \\ :\Leftrightarrow \quad S_{s_1} \leq_K S_{s_2} \wedge P_{s_1} \leq_A P_{s_2} \wedge O_{s_1} \leq_K O_{s_2}.$$

Die fachliche Semantik einer Aktivität eines Geschäftsprozessmodells spezifizieren wir, indem wir die Aktivität gemäß der in Abschnitt 8.3.1 vorgestellten Rekonstruktion als normsprachlichen Satz formulieren. Eine solche Spezifikation einer betrieblichen Aktivität stellt informell ausgedrückt die Aufforderung an die Umgebung dar, den beschriebenen Dienst zu erbringen bzw. auszuführen. Dabei erfolgt die Spezifikation einer Aktivität so allgemein, wie es die Anforderungen des Prozesses zulassen. Formal können wir die fachliche Semantik einer Aktivität folgendermaßen definieren:

Definition 8.3 (Fachliche Semantik einer Aktivität) *Seien $T = (T_K, T_A, \leq_K, \leq_A)$ eine Terminologie und a eine Aktivität eines Geschäftsprozessmodells, die durch den Satz $s_a \in T$ spezifiziert ist. Dann ist die fachliche Semantik $\mathcal{S}_{Akt}$ von a gegeben durch:*

$$\mathcal{S}_{Akt}(a) := s_a$$

Wir integrieren unseren Ansatz zur Spezifikation der fachlichen Semantik in lineare Prozessmodelle, indem wir in Ermangelung graphischer Repräsentationen für Organisationseinheiten und Ressourcen den gewählten normsprachlichen Satz als Bezeichner der Aktivität verwenden. Abbildung 8.12 zeigt das aus Abbildung 8.3 bekannte Geschäftsprozessmodell als lineares Prozessmodell mit normsprachlich spezifizierten Aktivitäten.

Analog zur Spezifikation der fachlichen Semantik einer Aktivität geben wir die Semantik einer Operation an, indem wir die Operation mit normsprachlichen Sätzen assoziieren, die sich an der in Abschnitt 8.3.1 vorgenommenen Rekonstruktion orientieren. Informell ausgedrückt beschreibt die Zuordnung eines Satzes zu einer Operation die Fähigkeit der Operation, den normsprachlich spezifizierten *und* jeden spezielleren Dienst erbringen zu können. Um die möglicherweise verschiedenen, mit der Operation verknüpften Fähigkeiten beschreiben zu können, ist es im Unterschied zur Spezifikation der fachlichen Semantik von Aktivitäten zulässig, einer Operation mehrere Sätze zuzuordnen. Die fachliche Semantik einer Operation ist formal wie folgt definiert:

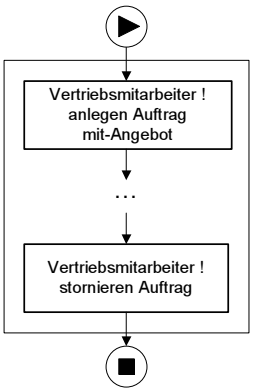


Abbildung 8.12: Lineares Prozessmodell mit normsprachlich spezifizierten Aktivitäten

Definition 8.4 (Fachliche Semantik einer Operation) *Seien* $T=(T_K, T_A, \leq_K, \leq_A)$ *eine Terminologie und o eine Operation einer Komponente, die durch Sätze* $s_{o,i} \in T$, $1 \leq i \leq n$ *spezifiziert ist. Dann ist die fachliche Semantik* $\mathcal{S}_{Op}$ *von o gegeben durch:*

$$\mathcal{S}_{Op}(o) := \bigcup_{1 \leq i \leq n} \{\, s \in T \mid s \leq_T s_{o,i} \,\}$$

Wir erweitern die in Abschnitt 6.1.3 eingeführte Syntax der CDL nun um die Möglichkeit, Operationen normsprachliche Sätze zuzuordnen. Listing 8.1 skizziert die um normsprachliche Annotationen ergänzte CDL-Beschreibung einer Komponente, die auf den bereits in Abbildung 8.2 eingeführten Klassen basiert.

```
component OrderManagement {
    class Offer {
        ...
    };
    class Order {
        ...
    };

    createOrder(in offer: Offer, out order: Order)
        [ Vertriebsangestellter anlegen Auftrag mit-Angebot ] {
        ...
    };
    delOrder(in order: Order)
        [ Vertriebsangestellter löschen Auftrag ] {
        ...
    };

    created()     = initialized();
    initialized() = ...
};
```

Listing 8.1: CDL-Komponentenbeschreibung mit normsprachlichen Annotationen

In Abschnitt 7.5 haben wir eine *abstrakte* Relation *Impl* eingeführt, die Aktivitäten solche Operationen zuordnet, die eine geeignete Implementierung der Aktivität versprechen (vgl. Definition 7.12). Mit unserem Wissen um die fachliche Semantik von Aktivitäten und Operationen sind wir nun in der Lage, dieser Relation eine *konkrete* Definition zu geben. Dem Subtyping-Prinzip folgend verlangen wir, dass eine Operation als Implementierung einer Aktivität in der Lage ist, den durch die Aktivität spezifizierten oder aber einen spezielleren Dienst zu erbringen. Bei der Definition der Relation *Impl* sind zwei Fälle zu unterscheiden, die in Abbildung 8.13 anhand einer Spezialisierungsrelation zwischen normsprachlichen Sätzen zur Spezifikation der fachlichen Semantik einer Aktivität (s_1 und s_3) und zur Beschreibung der fachlichen Semantik einer Operation (s_2) illustriert werden:

1. Operation spezieller als Aktivität ($s_2 \leq s_1$): Dieser Fall entspricht direkt unserer Anforderung, dass eine Operation den durch die Aktivität spezifizierten oder einen spezielleren Dienst erbringt.

2. Aktivität spezieller als Operation ($s_3 \leq s_2$): Nach Definition der fachlichen Semantik von Operationen unterstützt die Operation neben dem durch den Satz s_2 spezifizierten Dienst auch alle Dienste, die spezieller als s_2 sind. Folglich implementiert die Operation auch den Dienst s_3.

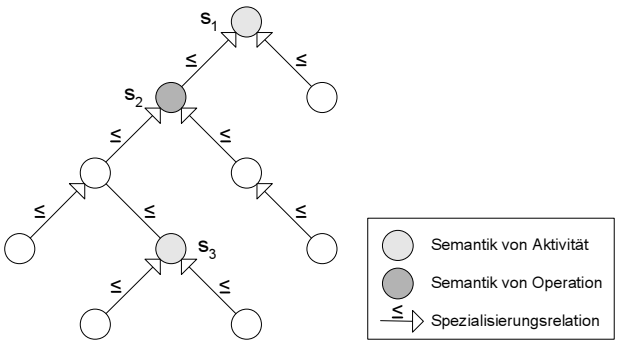


Abbildung 8.13: Spezialisierungsbeziehungen zwischen normsprachlichen Sätzen

Auf der Grundlage dieser Überlegungen können wir nun die Relation *Impl* für den Vergleich auf der Semantikebene folgendermaßen definieren:

Definition 8.5 (Implementierungsrelation (konkret)) *Seien* $T = (T_K, T_A, \leq_K, \leq_A)$ *eine Terminologie, a eine Aktivität und o eine Operation. Dann gilt:*

$$(a, o) \in Impl \text{ bzw. } o \in Impl(a) :\Leftrightarrow \exists s \in \mathcal{S}_{Op}(o) \bullet s \leq_T \mathcal{S}_{Akt}(a)$$

Aus dieser Definition folgt logisch, dass die Implementierungsbeziehung zwischen Aktivität und Operation unabhängig von der Richtung der Spezialisierungsbeziehung zwischen normsprachlichen Sätzen zur Beschreibung von Aktivität und Operation ist. Eine Operation implementiert genau dann eine Aktivität, wenn der ihre fachliche Semantik beschreibende Satz spezieller *oder* genereller als der Satz ist, der die fachliche Semantik der Aktivität spezifiziert.

8.5 Zusammenfassung

In diesem Kapitel haben wir unsere Konzepte für den semantischen Vergleich von Geschäftsprozessmodellen und Komponentenbeschreibungen vorgestellt. Die Einführung dieses Vergleichs auf der Semantikebene ist aus didaktischen Erwägungen als zweiter Schritt der Suchphase erfolgt. Tatsächlich muss aber der semantische Vergleich als Teil des Protokollvergleichs erfolgen oder diesem gar vorangehen (vgl. hierzu auch die Bezüge zur Implementierungsrelation in Definition 7.15).

Eingangs dieses Kapitels haben wir zunächst zustandsorientierte und vorgangsorientierte Alternativen zur Spezifikation der fachlichen Semantik von Operationen und betrieblichen Aktivitäten betrachtet. Mit Blick auf die Einbeziehung von Fachexperten in die Suche und Auswahl von Komponenten haben wir uns für einen vorgangsorientierten Ansatz entschieden, der Spezifikationen fachlicher Semantik mit (natürlich-)sprachlichem Charakter ermöglicht. Grundlage unseres Ansatzes zur vorgangsorientierten Spezifikation von Operationen und Aktivitäten stellen Normsprachen dar, die in die Kategorie der kontrollierten Sprachen einzuordnen sind und durch methodische Rekonstruktion der in einem Anwendungsbereich gesprochenen (Fach-)Sprache entwickelt werden. Wir haben eine normsprachliche Repräsentation der fachlichen Semantik von Aktivitäten und Operationen vorgeschlagen, indem wir zunächst die aus Sicht der geschäftsprozessorientierten Komponentensuche relevanten Eigenschaften von Geschäftsprozessmodellen und Komponentenbeschreibungen normsprachlich rekonstruiert und anschließend ein terminologisches Modell für die rechnergestützte Darstellung der resultierenden Normsprache entwickelt haben. Auf der Grundlage dieser Normsprache haben wir schließlich die fachliche Semantik von Aktivitäten und Operationen formal definiert, die wir für den semantischen Vergleich von Geschäftsprozessmodellen und Komponentenbeschreibungen heranziehen.

Der von uns in diesem Kapitel vorgestellte Ansatz zur Spezifikation der fachlichen Semantik weist interessante Parallelen zu der Beschreibung von Funktionen im Softwareinformationssystem LaSSIE (vgl. Abschnitt 4.3.2) auf. Vergleichbar mit unserem terminologischen Modell verwaltet LaSSIE Taxonomien von *Objects* (→ Konzepte), *Doers* (→ Organisationseinheiten) und *Actions* (→ Aktivitäten) sowie zusätzlich *States*, die in unserem Modell keine Entsprechung finden. Für die Beschreibung von Funktionen der Komponenten eines Informationssystems stehen eine Menge vorgegebener Frames zur Verfügung, die ähnlich unseren Satzbauplänen die Kombination von *Objects*, *Doers*, *Actions* und *States* zur Spezifikation von Funktionen reglementieren. Konzeptionelle Ähnlichkeit mit unserer Implementierungsrelation *Impl* ist zudem dadurch gegeben, dass in LaSSIE Funktionen gesucht werden, indem die Individuen derjenigen Frames bestimmt werden, die spezieller als die Anfrage sind.

Ein interessantes Konzept für die Erweiterung der Komponentensuche auf der Semantikebene stellt das *Action Refinement* [Weh02] dar, das Verfeinerungsbeziehungen zwischen Aktionen auf verschiedenen Abstraktionsebenen betrachtet. Dabei wird davon ausgegangen, dass atomare Aktionen einer abstrakten Ebene in eine Vielzahl von Aktionen auf einer konkreteren Ebene zerfallen. Im Kontext der geschäftsprozessorientierten Komponentensuche könnte das Action Refinement eingesetzt werden, um einer betriebli-

chen Aktivität eine Menge von Operationen zuzuordnen, deren Kombination eine Implementierung der geforderten Aktivität gestattet.

Der Aufbau organisationsweit oder gar organisationsübergreifend geltender Normsprachen und der mit ihnen verbundenen Terminologien ist zweifelsohne mit nicht unerheblichem Aufwand verbunden. Dass dieser Ansatz jedoch zumindest für abgegrenzte Anwendungsbereiche gangbar ist, haben die praktischen Erfahrungen von Ortner sowie das vom FORWIN entwickelte *ICF-System (Industries, Characteristics, Functions)* [KLM01] gezeigt. Das ICF-System definiert in einer Terminologie eine umfassende Hierarchie betrieblicher Funktionen, die für die Anforderungsanalyse genutzt wird. Folgeversionen dieser Terminologie werden gegenwärtig von der DATEV in einem Komponenten-Repository [Hau01] eingesetzt. Studer et al. haben mit dem *Corporate History Analyzer (CHARS)* [SSSS01] eine ontologiebasierte Anwendung für die Verwaltung und Analyse von Unternehmenshistorien realisiert. Im Rahmen dieser Arbeit wurde ein Metaprozess zur Unterstützung des Aufbaus und der Pflege von Ontologien entwickelt.

Teil III

Entwurf, Implementierung und Evaluation

Kapitel 9

Entwurf und Implementierung

Die in Teil II dieser Arbeit vorgestellten Konzepte sind im Rahmen des OFFIS[1]-Projekts *KOSOBAR (Komponentenbasierte Softwareentwicklung auf Basis von Referenzmodellen)* prototypisch umgesetzt worden. In diesem Kapitel stellen wir ausgewählte Aspekte des Entwurfs und der Implementierung der entstandenen Werkzeuge vor. Abschnitt 9.1 beschreibt zunächst eine abstrakte Systemarchitektur und dokumentiert anschließend den Entwurf der einzelnen Werkzeuge. Die Verfeinerung der Architektur sowie der Werkzeuge im Zuge der Implementierung wird in Abschnitt 9.2 dargestellt. Abschnitt 9.3 fasst das Kapitel zusammen.

9.1 Entwurf

Ausgehend von der Identifikation relevanter Anwendungsfälle für das in dieser Arbeit betrachtete Szenario führen wir in diesem Abschnitt zunächst eine abstrakte Systemarchitektur ein, die einzelne Werkzeuge des Entwurfs identifiziert sowie deren Abhängigkeiten überblicksartig beschreibt. Im Anschluss gehen wir auf ausgewählte Aspekte des Entwurfs einiger Werkzeuge näher ein.

9.1.1 Systemarchitektur

Die Entwicklung der Systemarchitektur für die prototypische Realisierung der in dieser Arbeit vorgestellten Konzepte war auf die Berücksichtigung der spezifischen Aufgaben ausgerichtet, die wir den Akteuren im Makroprozess geschäftsprozessorientierter Komponentensuche in Abschnitt 5.2 zugeordnet haben. Das in Abbildung 9.1 dargestellte Anwendungsfalldiagramm ordnet den Akteuren „Komponentenhersteller“, „Normierungsgremium“ und „Komponentenverwender“ zentrale funktionale Anforderungen an das zu realisierende Gesamtsystem in Form von Anwendungsfällen zu. So benötigt ein Komponentenhersteller eine Möglichkeit, die von ihm entwickelten Komponenten auf einem Komponentenmarkt anzubieten, indem er ihre Beschreibungen veröffentlicht. Die Aufgaben eines Normierungsgremiums sind durch geeignete Funktionalität zur Verwaltung

[1] OFFIS (Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme, http://www.offis.de/) ist ein so genanntes An-Institut der Carl von Ossietzky Universität Oldenburg.

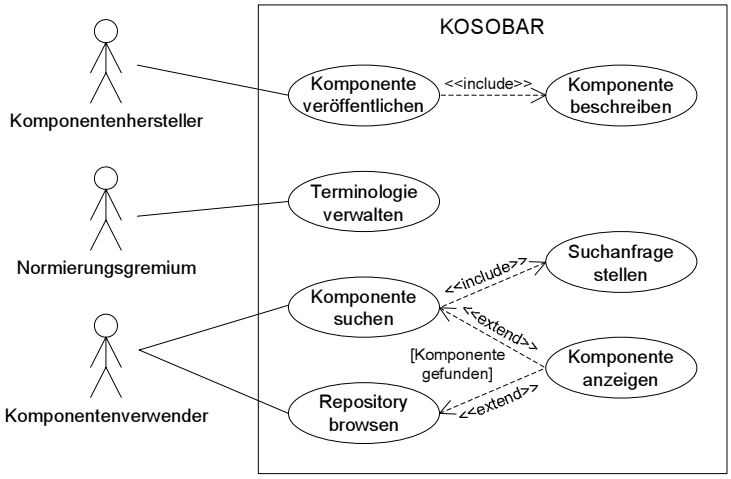


Abbildung 9.1: Anwendungsfalldiagramm mit KOSOBAR-Kernfunktionen

von Terminologien zu unterstützen. Der zentrale Anwendungsfall mit Beteiligung eines Komponentenverwenders ist die geschäftsprozessorientierte Suche nach Komponenten. Als Teil dieser Suchfunktionalität benötigt ein Verwender eine Möglichkeit, Suchanfragen in Gestalt von Geschäftsprozessmodellen an das System zu stellen. Sofern geeignete Komponenten gefunden wurden, müssen diese dem Komponentenverwender in angemessener Form präsentiert werden können. Letztere Funktionalität soll dem Komponentenverwender auch unabhängig von der Komponentensuche (z. B. beim Browsen durch den Bestand aller angebotenen Komponenten) zur Verfügung stehen. Da die Akteure „Broker“ und „Marktplatzbetreiber“ als Anbieter dieser Kernfunktionalitäten auftreten, werden sie im Diagramm nicht aufgeführt.

Aus diesen zentralen Anwendungsfällen haben wir eine abstrakte Systemarchitektur abgeleitet, die in Abbildung 9.2 illustriert ist und die Anforderungen von Komponentenhersteller, Komponentenverwender und Normierungsgremium durch Werkzeuge zur Komponentenverwaltung, Komponentensuche und Terminologieverwaltung erfüllt. Mit Ausnahme des Aufgabenbereichs der Komponentensuche, in dem keine persistente Datenhaltung erforderlich ist, sind diese Werkzeuge in einer 3-schichtigen Client/Server-Architektur mit der clientseitigen Schicht „Front-End-Werkzeuge“ und den serverseitigen Schichten „Anwendungsdienste“ und „Datenhaltung“ angeordnet. Der Zugriff der Front-End-Werkzeuge auf die Werkzeuge des Back-End-Bereichs soll dabei über das Internet erfolgen können.

Ein Komponentenhersteller setzt zur Verwaltung von Komponenten bzw. deren Beschreibungen mit dem *CDL Component Manager* ein Werkzeug ein, das auf dem *CDL Component Server* aufsetzt. Der CDL Component Server definiert eine Schnittstelle, die die Verwaltung von Komponenten gemäß dem durch die CDL definierten Beschreibungsumfang gestattet. Für die persistente Speicherung von CDL-Komponentenbeschreibungen bzw. deren Veröffentlichung auf einem Komponentenmarkt steht das *CDL Component Repository* zur Verfügung. Vergleichbare Strukturen finden sich im Aufgabenbereich der Terminologieverwaltung wieder. Hier unterstützt der *Terminology Manager* das Nor-

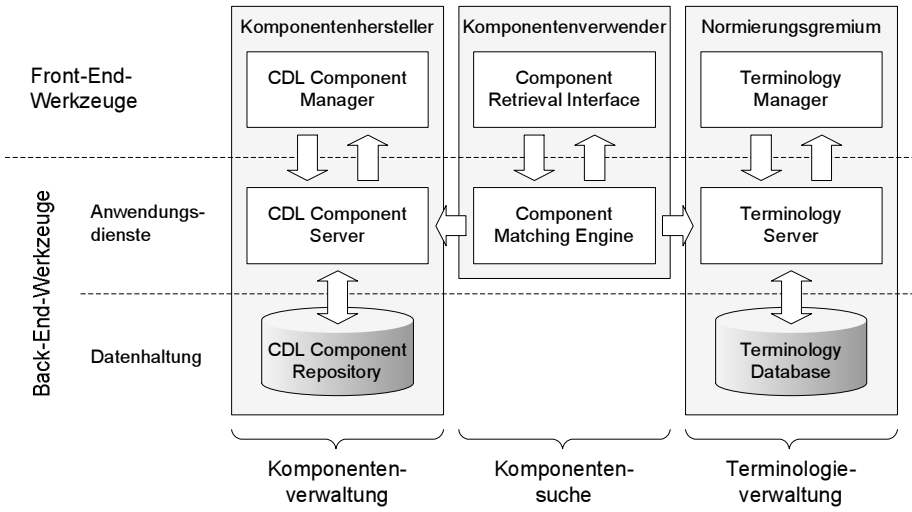


Abbildung 9.2: Architekturübersicht (abstrakt)

mierungsgremium bei der Verwaltung von Terminologien. Serverseitig werden Dienste für den Zugriff auf eine Terminologie von einem *Terminology Server* angeboten, der für die persistente Datenhaltung auf eine *Terminology Database* zurückgreift. Neben dem Terminology Manager greift auch der CDL Component Manager zwecks Definition der fachlichen Semantik von Operationen auf den Terminology Server zu. Diese Abhängigkeit ist in Abbildung 9.2 nicht dargestellt. Das *Component Retrieval Interface* unterstützt den Komponentenverwender bei der Komponentensuche, indem es neben der Formulierung von Suchanfragen auch die Präsentation von (gefundenen) Komponenten ermöglicht. Die *Component Matching Engine* führt die eigentliche geschäftsprozessorientierte Komponentensuche durch und nutzt dazu die Dienste des CDL Component Servers sowie des Terminology Servers.

9.1.2 Komponentenverwaltung

In diesem Abschnitt gehen wir näher auf ausgewählte Aspekte der Werkzeuge zur Komponentenverwaltung ein. Da aus konzeptioneller Sicht nur der CDL Component Server sowie das darunter liegende CDL Component Repository interessant sind, verzichten wir auf eine Darstellung von Entwurfsdetails zum CDL Component Manager.

CDL Component Server Aufgabe des CDL Component Servers ist die Bereitstellung einer Programmierschnittstelle (API, Application Programming Interface) für die Verwaltung von Komponenten bzw. deren CDL-Beschreibungen. Dieses API baut auf dem CDL-Komponentenmodell (vgl. Abschnitt 6.1.2) auf und ergänzt Strukturen für die Beschreibung des Verhaltens von Komponenten. Abbildung 9.3 zeigt ein Klassendiagramm, das einen groben Überblick über das API des CDL Component Servers (kurz CDL-API) gibt.

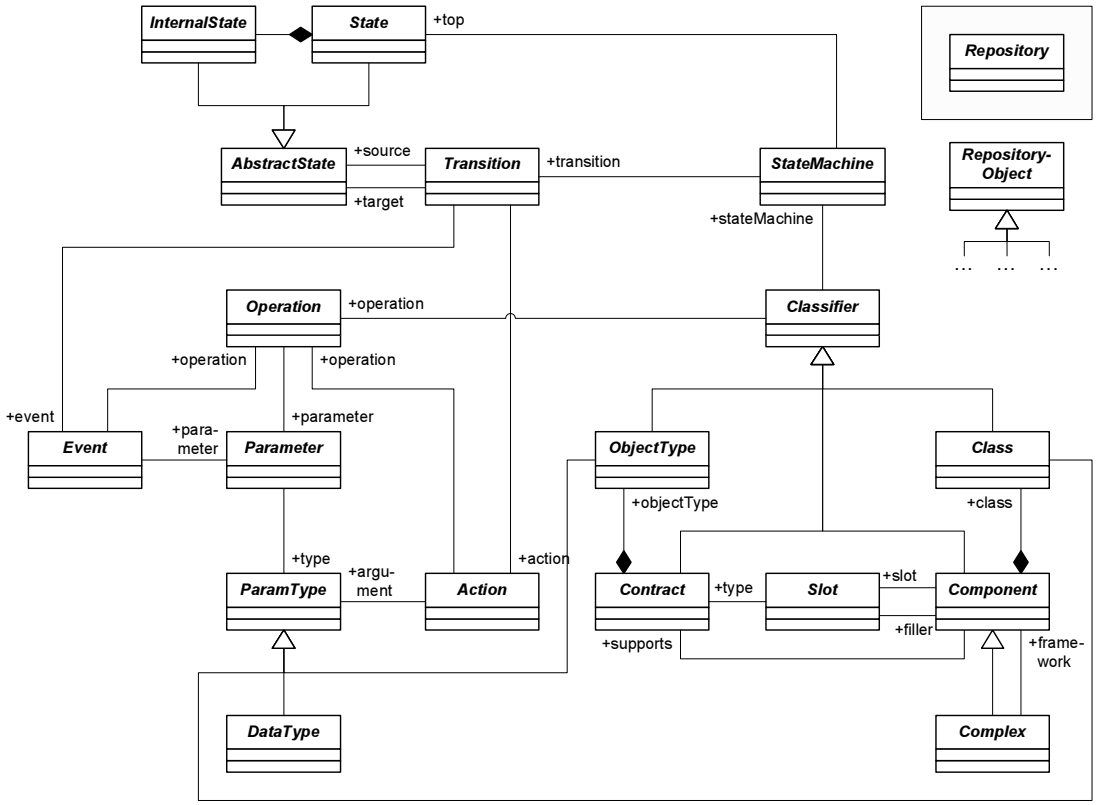


Abbildung 9.3: API des CDL Component Servers

Die Klasse `RepositoryObject` ist die Basisklasse für alle Objekte, die zur Repräsentation einer Komponente vom CDL Component Manager im CDL Component Repository persistent abgelegt werden. `RepositoryObject`s werden durch eine Instanz der Klasse `Repository` verwaltet, die den zentralen Ausgangspunkt für den Zugriff auf Komponentenbeschreibungen darstellt. Neben Methoden für die Suche nach `RepositoryObject`s bietet die Klasse `Repository` auch Methoden zur Erzeugung von (komplexen) Komponenten und Kontrakten an. Dieses als *Fabrikmethode* [GHJV01] bekannte Entwurfsmuster wird im CDL-API durchgängig verwendet.

Grundlage für die Beschreibung der strukturellen Einheiten `Component`, `Contract`, `Class` und `ObjectType` ist die Klasse `Classifier`, mit deren Hilfe sich die in einer Schnittstelle zusammengefassten `Operation`s verwalten lassen. Einem `Operation`-Objekt können `Parameter`, der CDL-Code des Operationsrumpfes sowie eine fachliche Semantik in Gestalt von Referenzen auf normsprachliche Aussagen zugeordnet werden. Die Klassen `Component` und `Contract` gestatten darüber hinaus die Beschreibung der indirekten Schnittstellen von Komponenten und Kontrakten durch Methoden für die Verknüpfung mit angebotenen `Class`es bzw. erwarteten `ObjectType`s. Zudem lassen sich die Abhängigkeiten einer Komponente von anderen Komponenten mittels `Slots` verwalten, deren Typ durch den zugeordneten `Contract` definiert ist. Die von einer Komponente unterstützten Kontrakte lassen sich spezifizieren, indem einer `Component` die entsprechenden

`Contract`-Objekte zugeordnet werden. Über die Klasse `Complex` können komplexe Komponenten verwaltet werden. Sie gestattet die Spezifikation der `Component`, die als der komplexen Komponente zugrunde liegendes Framework agiert. Ein `Slot` einer komplexen Komponente kann mit `Component`s parametrisiert werden.

Das Verhalten eines `Classifier`s ist durch dessen Verknüpfung mit einem Objekt der Klasse `StateMachine` definiert. Eine `StateMachine` ist aus `Transition`s zwischen Zuständen aufgebaut, die durch die abstrakte Oberklasse `AbstractState` repräsentiert werden. Ein Zustand kann dabei entweder ein `State` oder aber als `InternalState` Teil eines `State`s sein. `InternalState`s lassen sich einsetzen, um komplexere Ausdrücke zur Beschreibung des in einem Zustand möglichen Verhaltens abzubilden. Das aktive Verhalten, das mit dem `created`-Zustand einer CDL-Komponentenbeschreibung verbunden sein kann, lässt sich durch die Verknüpfung einer `Transition` mit einer `Action` repräsentieren. Einer solchen `Action` können neben der auszuführenden `Operation` die Namen und Typen der Argumente zugeordnet werden, mit denen diese ausgeführt wird. Die Spezifikation des passiven Verhaltens, das in den übrigen Zuständen möglich ist, wird durch die Verknüpfung einer `Transition` mit `Event`s erreicht. Einem `Event` sind die erwartete `Operation` sowie die `Parameter` zugeordnet, mit denen die Operation aufgerufen wird. Zulässige Typen eines Parameters sind durch die Klasse `ParamType` definiert, von der neben `ObjectType` und `Class` auch die Klasse `DataType` abgeleitet ist. Ein `DataType` repräsentiert einen Basisdatentypen wie z. B. `String` oder `int`.

CDL Component Repository Das CDL Component Repository dient der persistenten Speicherung von Komponentenbeschreibungen. Als Speicherungsstruktur haben wir das Metamodell der UML gewählt, da dieses als Standard der OMG eine hohe Verbreitung gefunden hat und von vielen Werkzeugen (z. B. Modellierungswerkzeugen und Repositories) unterstützt wird. Im Folgenden gehen wir auf die zentralen Aspekte der Abbildung des CDL-APIs auf das Metamodell der UML näher ein.

Abbildung 9.4 stellt die Zuordnung der strukturellen Aspekte des CDL-APIs zum Metamodell der UML graphisch dar. Dabei beschreibt das Klassendiagramm den von uns genutzten Ausschnitt des UML-Metamodells, während die grauen Hinterlegungen auf die durch die UML-Elemente repräsentierten CDL-Konstrukte hinweisen. In dem Klassendiagramm wird die Klasse `Component` mehrfach dargestellt, um eine höhere Übersichtlichkeit zu erzielen. Sie ist gleichermaßen Grundlage unserer Repräsentation von Komponenten, Kontrakten und komplexen Komponenten. Zur Unterscheidung des Kontexts, in dem ein `Component`-Objekt verwendet wird, setzen wir den UML-Erweiterungsmechanismus *Stereotype* ein. Über ein entsprechendes Objekt der Klasse `Stereotype` ordnen wir der Basisklasse `Classifier` (und damit der `Component`) eine Typisierung als Komponente, Kontrakt oder eben komplexe Komponente zu. Einem `Classifier`, auf dessen Grundlage wir indirekt auch Objekttypen, Klassen und Basisdatentypen repräsentieren, können `Operation`s zugeordnet sein, deren Parameter wiederum mit `Classifier`-Objekten typisiert sind. Die Verknüpfung einer Operation mit normsprachlichen Aussagen aus einer Terminologie wird durch den UML-Erweiterungsmechanismus *Tagged Value* erreicht, mit dessen Hilfe Modellelemente um Attribut/Wert-Paare ergänzt werden können. Der Einsatz der Klasse `TaggedValue` zu diesem Zweck ist in Abbildung 9.4 zugunsten einer höheren Übersichtlichkeit nicht dargestellt.

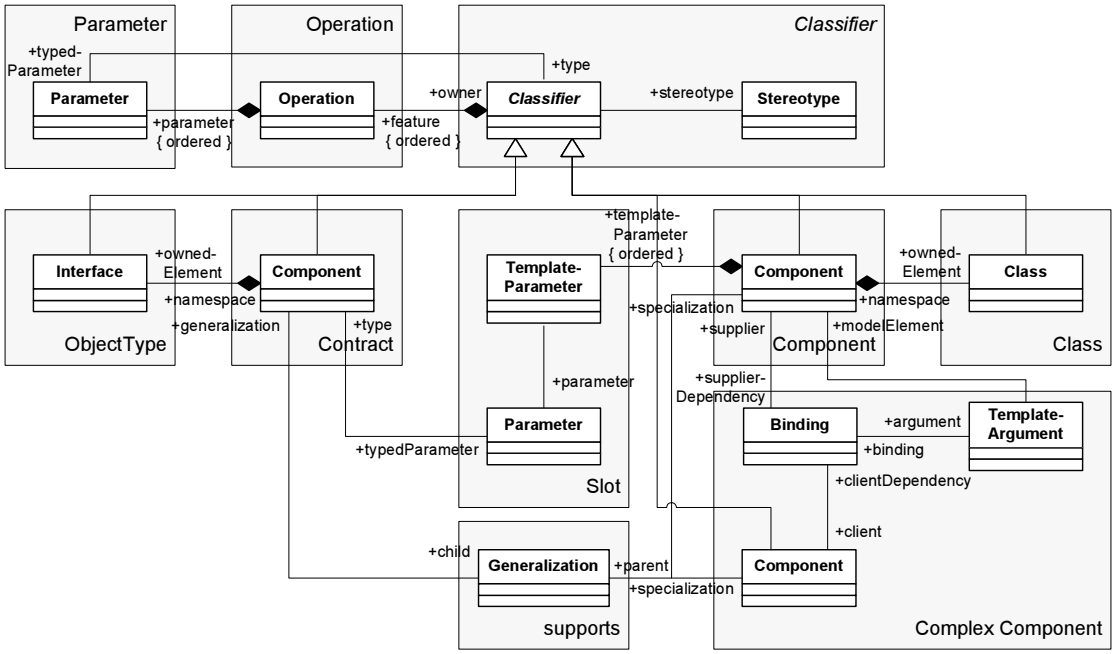


Abbildung 9.4: Abbildung des CDL-APIs auf das UML-Metamodell (Struktur)

Zur Abbildung der Unterstützung eines Kontrakts durch eine Komponente verbinden wir die jeweiligen `Component`s über ein Objekt der Klasse `Generalization`, die eine Generalisierungsbeziehung zwischen einer (spezielleren) Komponente und einem (generelleren) Kontrakt modelliert. Slots einer Komponente werden repräsentiert, indem eine `Component` eine Reihe (formaler) `Parameter` definiert, die mit Kontrakten (`Component`s) typisiert sind. Die Parametrisierung einer komplexen Komponente durch die Belegung der Slots ihres Komponentenframeworks mit Komponenten wird durch ein `Binding` beschrieben. Ein `Binding` verknüpft eine `Component` in der Rolle der komplexen Komponente (`client`) mit einer `Component` in der Rolle des Frameworks (`supplier`) sowie einer Reihe weiterer `Component`s, die die Belegung der Slots des Komponentenframeworks spezifizieren (`argument`).

Die Abbildung der dynamischen Anteile des CDL-APIs auf das Metamodell der UML ist in Abbildung 9.5 graphisch dargestellt. Grundlage der Beschreibung des Verhaltens eines `Classifier`s ist die Klasse `StateMachine`, die auf einen Anfangszustand und eine Menge von Transitionen verweist. Ein `State` aus dem CDL-API wird durch einen `CompositeState` der UML repräsentiert. Ein solcher `CompositeState` beinhaltet eine Automatenstruktur, die zumindest einen `PseudoState` als Anfangszustand umfasst. Diese Automatenstruktur kann durch `SimpleState`-Objekte ergänzt werden, mit denen wir `InternalState`s des CDL-APIs repräsentieren. Transitionen zwischen Zuständen werden durch `Transition`s zwischen internen Zuständen dargestellt, d. h. zwischen `PseudoState`s und/oder `SimpleState`s. Die bedingte Ausführbarkeit einer Transition wird durch die Verknüpfung einer `Transition` mit einem `Guard` erreicht. Aktives Verhalten, wie es im Zustand `created` möglich ist, wird durch `CallAction`s repräsentiert, denen jeweils eine

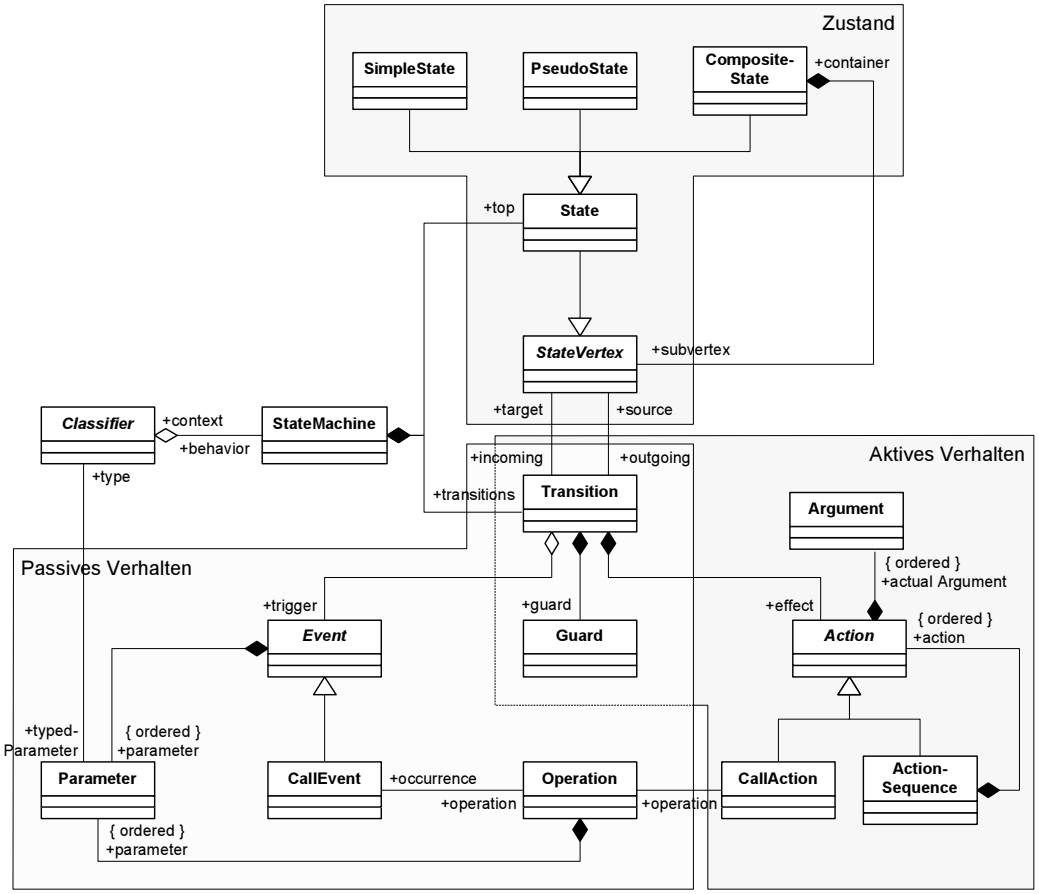


Abbildung 9.5: Abbildung des CDL-APIs auf das UML-Metamodell (Verhalten)

`Operation` und eine Reihe von `Argument`s zugeordnet ist. `CallAction`s können dabei nach dem *Kompositum*-Entwurfsmuster [GHJV01] zu komplexeren Verhaltensbeschreibungen kombiniert werden. Die Beschreibung des passiven Verhaltens, das den übrigen Zuständen zugeordnet sein kann, wird auf ähnliche Weise durch `CallEvent`s erreicht, die wiederum mit einer aufrufbaren `Operation` und einer Reihe von `Parameter`s assoziiert sind.

9.1.3 Terminologieverwaltung

Ziel dieses Abschnitts ist die Erläuterung relevanter Entwurfsentscheidungen hinsichtlich der Werkzeuge zur Terminologieverwaltung. Da der Terminology Manager als Benutzungsschnittstelle konzeptionell nicht interessant erscheint, verzichten wir auf die Vorstellung seines Entwurfs. Ebenso verzichten wir auf die Darstellung von Details zum (Schema-)Entwurf der Terminology Database, da wir für die Realisierung des Terminology Servers auf eine Fremdkomponente zugreifen, die die Abbildung konzeptioneller Strukturen auf ein relationales Datenbankschema vornimmt. Auf den Einsatz dieser Fremdkomponente gehen wir im Rahmen der Dokumentation der Implementierung in Abschnitt 9.2.2 näher ein.

Terminology Server Dem Terminology Server fällt die Aufgabe zu, eine Programmierschnittstelle für die Verwaltung von Terminologien gemäß der in Kapitel 8 aufgestellten Anforderungen bereitzustellen. Abbildung 9.6 stellt das durch einen Terminology Server zu implementierende API zur Verwaltung von Terminologien in Form eines Klassendiagramms überblicksartig dar.

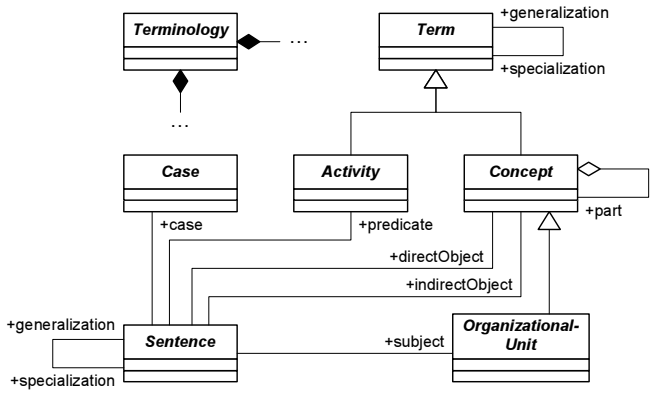


Abbildung 9.6: API des Terminology Servers

Das API des Terminology Servers leitet sich direkt aus dem in Abbildung 8.11 dargestellten terminologischen Modell ab. Kleinere Abweichungen ergeben sich lediglich dadurch, dass im API nicht zwischen unterschiedlichen Kopulae unterschieden wird und dass das API nicht die Beschreibung von Ganzes/Teile-Beziehungen zwischen Aktivitäten gestattet. Die Unterscheidung der für die Formulierung von Aktivitäten eines Geschäftsprozessmodells bzw. der fachlichen Semantik von Operationen einer Komponente genutzten Kopulae ist für die geschäftsprozessorientierte Komponentensuche irrelevant und kann damit in dem angestrebten Prototypen entfallen. Wie bereits in Abschnitt 8.3.2 angedeutet dienen Ganzes/Teile-Beziehungen lediglich der übersichtlicheren Strukturierung einer Terminologie. Sie können damit in einem Prototypen ebenfalls vernachlässigt werden, ohne die Evaluation der vorgestellten Konzepte negativ zu beeinflussen.

Vergleichbar mit der Rolle der Klasse `Repository` im CDL Component Server dient beim Terminology Server die Klasse `Terminology` als zentraler Einstiegspunkt in eine Terminologie. Neben Methoden für die Bestimmung der Wurzelknoten der in einer Terminologie verwalteten Konzept-, Organisationseinheits- und Aktivitätshierarchien sowie die Suche nach Termen und Sätzen in der Terminologie bietet diese Klasse *Fabrikmethoden* zur Erzeugung von Termen und Sätzen an.

9.1.4 Komponentensuche

Die Werkzeuge zur Komponentensuche stellen den Kern der Implementierung dar. Da das Component Retrieval Interface lediglich eine Benutzungsschnittstelle darstellt, über die Suchanfragen gestellt und gefundene bzw. verfügbare Komponenten angezeigt werden können, verzichten wir auf die Dokumentation seines Entwurfs und konzentrieren uns auf die Vorstellung des Entwurfs der Component Matching Engine.

Component Matching Engine Die Component Matching Engine realisiert die Suche nach geeigneten Komponenten auf der Grundlage von Geschäftsprozessmodellen. Die wichtigsten Aspekte ihres Entwurfs sind in Abbildung 9.7 als Klassendiagramm dargestellt. Dabei können die Bereiche „Service-To-Worker“, „Suchalgorithmus“, „Parser“ und „Transitionssysteme“ unterschieden werden, auf die wir im Folgenden genauer eingehen.

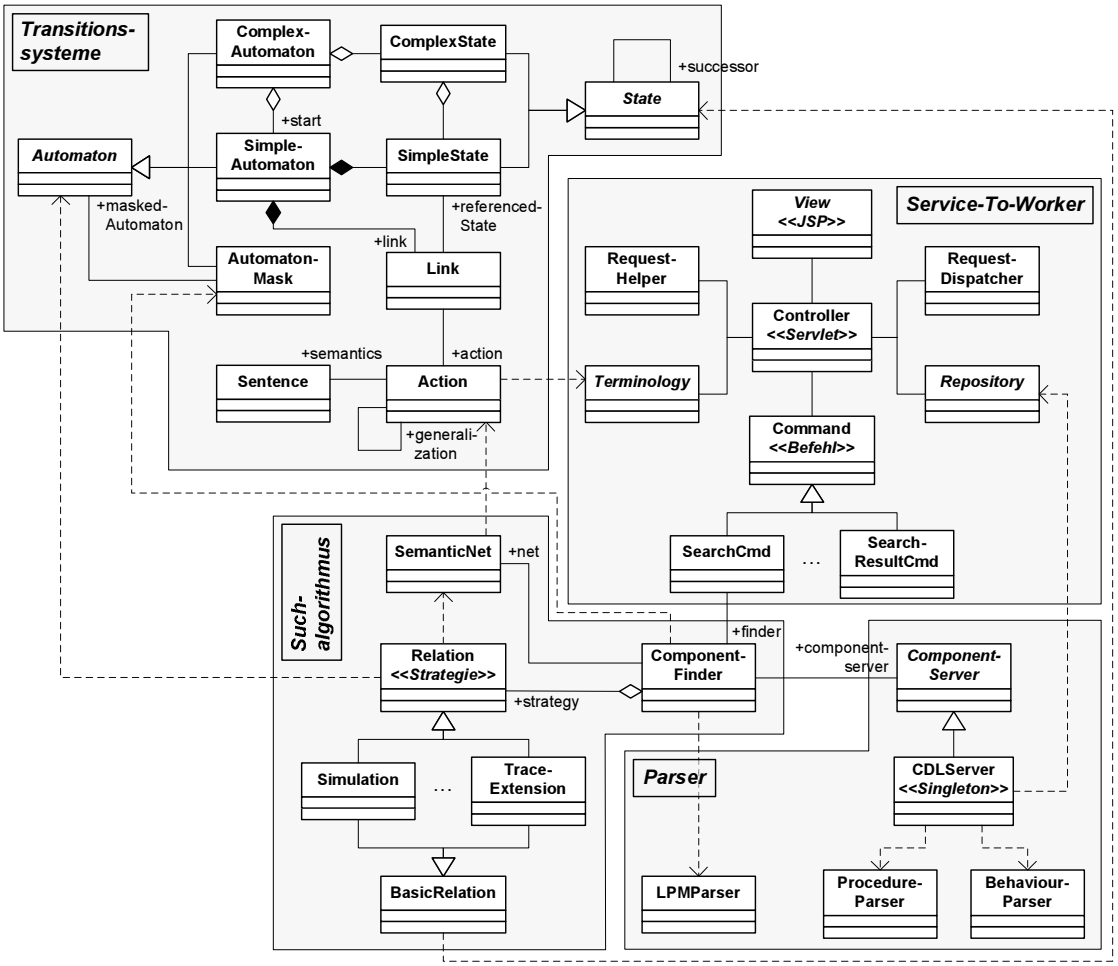


Abbildung 9.7: Überblick über den Entwurf der Component Matching Engine

Die Klassenstruktur des Bereichs „Service-To-Worker“ orientiert sich an dem gleichnamigen *J2EE-Entwurfsmuster* [Bie02] und definiert eine flexible Architektur für die Erstellung von Web-Anwendungen. Zentraler Zugangsknoten für jegliche Anfragen an die Component Matching Engine ist ein `Controller`, für den eine Realisierung durch ein *Servlet* [Hun01] vorgesehen ist. Der Controller wird mit Referenzen auf ein `Repository` sowie eine `Terminology` (vgl. Abschnitt 9.1.2 bzw. Abschnitt 9.1.3) initialisiert, die der Component Matching Engine den Zugriff auf verfügbare Komponenten sowie die verwendete Terminologie gestattet. Bei der Bearbeitung eingehender Anfragen unterstützt der

`RequestHelper` den Controller, indem er das auszuführende Kommando aus der Anfrage ermittelt. Ausführbare Kommandos wie z. B. `SearchCmd` und `SearchResultCmd` sind nach dem *Befehlsmuster* [GHJV01] von der abstrakten Oberklasse `Command` abgeleitet. Jedes Kommando liefert einen Rückgabewert, auf deren Grundlage der Controller die als nächstes darzustellende HTML-Seite auswählt. Die dynamische Generierung dieser HTML-Seiten erfolgt durch `Views`, die in Form von *JavaServer Pages (JSP)* [Ber02] realisiert und durch den `RequestDispatcher` angefordert werden.

Der Bereich „Suchalgorithmus“ umfasst die Klassen, die für die eigentliche Komponentensuche verantwortlich sind. Das bereits genannte Kommando `SearchCmd` übermittelt ein lineares Prozessmodell an den `ComponentFinder`, wobei eine XML-Repräsentation gemäß der Document Type Definition (DTD) aus Anhang B Verwendung findet. Der `ComponentFinder` überführt dieses XML-Dokument zunächst mit dem `LPMParser` aus dem Bereich „Parser“ in eine Repräsentation als Transitionssystem. Über den `CDLServer`, der eine *Singleton*-Implementierung [GHJV01] der abstrakten Klasse `ComponentServer` aus demselben Bereich darstellt, fordert der `ComponentFinder` sukzessive die Verhaltensbeschreibungen der im `Repository` verfügbaren Komponenten an, wobei für die interne Repräsentation des Verhaltens wiederum Transitionssysteme zum Einsatz kommen.

Für die Repräsentation von linearen Prozessmodellen und Verhaltensbeschreibungen stehen die Klassen des Bereichs „Transitionssysteme“ zur Verfügung. Ein `SimpleAutomaton` repräsentiert als Unterklasse der abstrakten Basisklasse `Automaton` ein einfaches Transitionssystem gemäß Definition 7.4, das aus einer Menge untereinander verbundener einfacher Zustände besteht. Einfache Zustände werden in der Component Matching Engine durch die Klasse `SimpleState` realisiert, wobei Informationen über die Transitionsrelation zwischen Zuständen in der abstrakten Oberklasse `State` verwaltet werden. Verweise eines einfachen Transitionssystems auf neu erzeugte oder existierende Instanzen einfacher Transitionssysteme (vgl. Definition 7.7) werden als `Link`-Objekte repräsentiert. Ein solcher Verweis referenziert neben dem `SimpleAutomaton`, dem er angehört, die `Action` einer Transition, durch deren Ausführung er aktiviert wird, sowie die Menge der aktivierten Zustände. Die fachliche Semantik von Objekten der Klasse `Action` ist durch `Sentence`-Objekte aus dem API des Terminology Servers (vgl. Abschnitt 9.1.3) definiert. Komplexe Transitionssysteme nach Definition 7.8, die aus den Verweisstrukturen zwischen einfachen Transitionssystemen resultieren, werden durch die Klasse `ComplexAutomaton` repräsentiert. Ein `ComplexAutomaton` besteht aus einem `SimpleAutomaton`, der den Ausgangspunkt für die dynamische Entwicklung des komplexen Transitionssystems nach Definition 7.11 definiert, und einer Menge komplexer Zustände (vgl. Definition 7.9), die Mengen alternativ aktiver `SimpleState`-Objekte zusammenfassen und durch die Klasse `ComplexState` abgebildet werden. Die Klasse `AutomatonMask` definiert als dritte Unterklasse von `Automaton` eine Sicht auf ein Transitionssystem, die eine zuvor ausgewählte Menge von `Action`s durch Hiding nach Definition 7.14 versteckt. Detaillierte Informationen bzgl. der Erzeugung der Transitionssysteme zur Repräsentation des Komponentenverhaltens durch den `BehaviourParser` sowie der Berechnung von Verweisen zwischen solchen Transitionssystemen durch den `ProcedureParser` finden sich in der Diplomarbeit von Keilers [Kei01]. In diesem Zusammenhang soll angemerkt werden, dass der `BehaviourParser` auf der textuellen Repräsentation der CDL aufsetzt, anstatt über die Klassen des CDL-API auf die Verhaltensbeschreibungen einer Komponente zuzugreifen.

Der eigentliche Suchprozess kann als zweistufiger (Komponenten-)Filter verstanden werden, bei dem zunächst die semantische Übereinstimmung zwischen angefragtem Prozessmodell und einer Komponente überprüft und erst bei Überschreitung eines vorzugebenden Schwellwertes der Vergleich der Protokolle vorgenommen wird. Der bereits eingeführte `ComponentFinder` berechnet dabei in einem ersten Schritt mittels der Klasse `SemanticNet` ein semantisches Netz, das unter Verwendung der Klasse `Action` die Generalisierungs-/Spezialisierungsbeziehungen zwischen den Aktivitäten des linearen Prozessmodells und den Operationen der betrachteten Komponente wiedergibt. Aktivitäten bzw. Operationen, für die es keine Generalisierung/Spezialisierung gibt, werden mittels einer `AutomatonMask` aus den jeweiligen Transitionssystemen ausgeblendet. Überschreitet der Anteil nicht versteckter Aktionen und damit die semantische Übereinstimmung zwischen den betrachteten (komplexen) Transitionssystemen einen Schwellwert, so wird der Protokollvergleich durchgeführt. Die dabei untersuchte binäre `Relation` zwischen Transitionssystemen bzw. deren Zuständen lässt sich gemäß dem *Strategie*-Muster [GHJV01] konfigurieren. Die konkrete Strategie `Simulation` implementiert den in Listing 7.1 vorgestellten Algorithmus zur Berechnung der Relation *Match*. Die Klasse `TraceExtension` weist im Klassendiagramm lediglich auf die Möglichkeit zur Verwendung anderer Vergleichsrelationen hin. Das Sequenzdiagramm in Abbildung 9.8 fasst die wesentlichen Interaktionen des beschriebenen Suchprozesses graphisch zusammen.

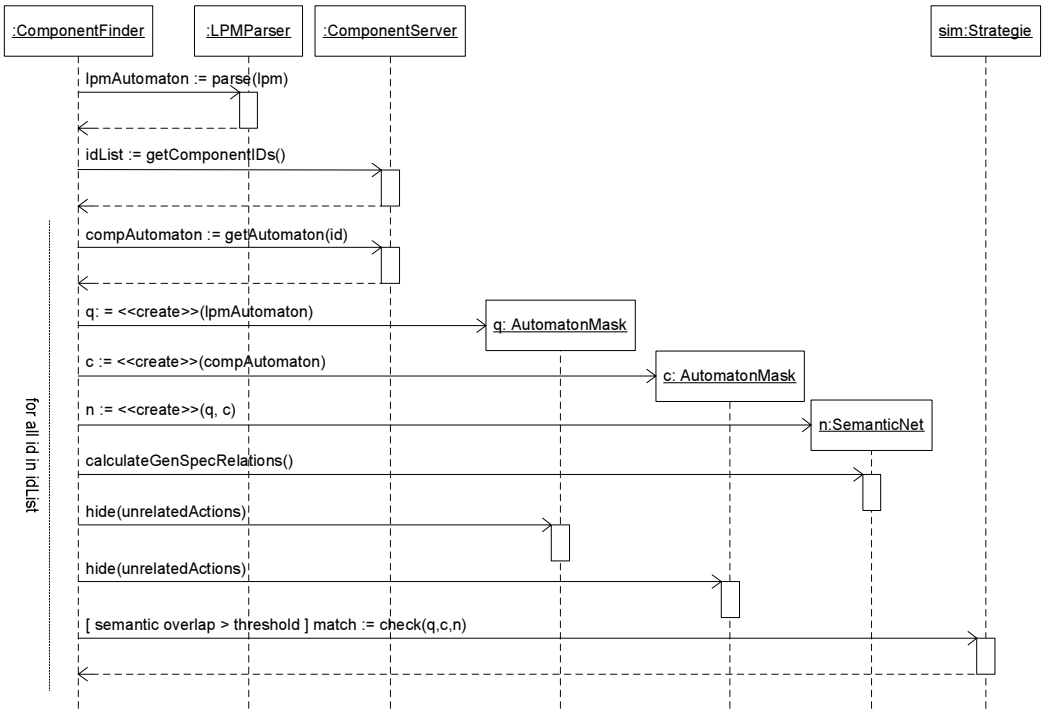


Abbildung 9.8: Zusammenfassung der Interaktionen zur Komponentensuche

9.2 Implementierung

Im Rahmen der Implementierung der KOSOBAR-Werkzeuge ist die abstrakte Systemarchitektur aus Abbildung 9.2 verfeinert worden. Abbildung 9.9 fasst die resultierende konkrete Systemarchitektur graphisch zusammen.

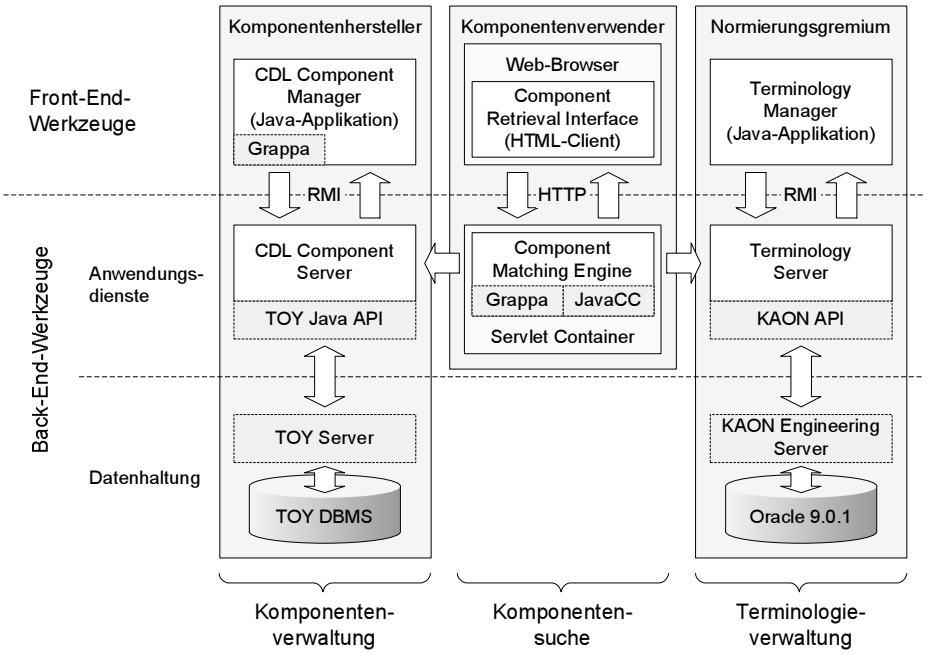


Abbildung 9.9: Architekturübersicht (konkret)

An der Entwicklung der Werkzeuge waren im Zeitraum von drei Jahren neben zwei wissenschaftlichen Mitarbeitern auch drei wissenschaftliche Hilfskräfte und ein Diplomand beteiligt. Einen Eindruck vom Umfang der Implementierung gibt Tabelle 9.1.

Paket	**Aufgabe**	**# Klassen**
de.offis.kosobar.broker.engine	Berechnung von *Match*	24
parser	Anfrage- und Komponentenparser	15
web	Service-To-Worker-Muster	23
de.offis.kosobar.repository.api	Schnittstelle	22
frontend	CDL Component Manager	32
toyrepository	CDL Component Server (TOY)	26
de.offis.kosobar.terminology.api	Schnittstelle	14
frontend	Terminology Manager	22
kaonserver	Terminology Server (KAON)	15

Tabelle 9.1: Umfang der Implementierung

In den folgenden Abschnitten erläutern wir die Systemarchitektur hinsichtlich der Werkzeuge zur Verwaltung von Komponenten und Terminologien sowie zur Komponentensuche. Wir beschränken uns dabei auf die wesentlichen Implementierungsaspekte.

9.2.1 Komponentenverwaltung

Der CDL Component Manager ist in Form einer Java-Applikation realisiert worden. Als „Thin Client“ nimmt er lediglich Aufgaben wahr, die in direktem Zusammenhang mit der Benutzerinteraktion stehen, und delegiert die für die logische (und persistente) Verwaltung von Komponentenbeschreibungen erforderliche Anwendungslogik an den CDL Component Server. Für die Kommunikation zwischen CDL Component Manager und dem ebenfalls in Java implementierten CDL Component Server kommt dabei *RMI (Remote Method Invocation)* zum Einsatz. Abbildung 9.10 zeigt einen Screenshot des CDL Component Managers. Für die graphische Darstellung von Verhaltensbeschreibungen in Form von Zustandsmaschinen und deren Manipulation setzen wir die Java-Graphenbibliothek *Grappa* [ATT03] der AT&T Labs-Research ein.

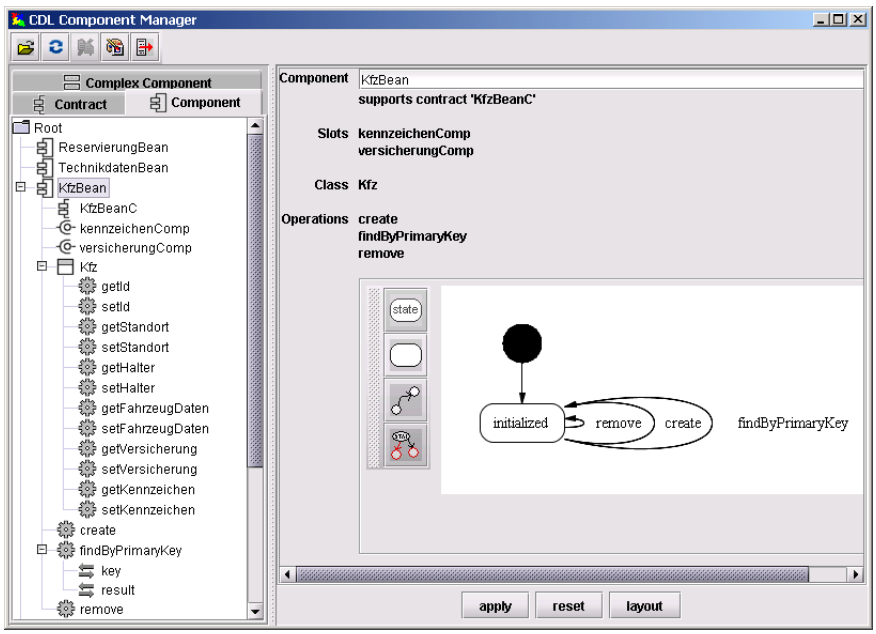


Abbildung 9.10: CDL Component Manager

Für die persistente Speicherung von Komponentenbeschreibungen im CDL Component Repository setzen wir das *TODAY Open Repository* (kurz TOY-Repository) [HT01] ein, das im Rahmen des OFFIS-Projekts *TODAY (Toolsuite for Managing a Data Warehouse's Data Supply)* entwickelt wurde. Das TOY-Repository gestattet das Anlegen von Metadatenstrukturen, die auf der *Meta Object Facility (MOF)* [OMG00a] der OMG basieren, sowie die Verwaltung entsprechender Metadaten. MOF definiert eine kompakte objektorientierte Sprache für die Spezifikation von Metadatenstrukturen, wie sie z. B. die UML darstellt. Die Initialisierung des TOY-Repository mit Metadatenstrukturen kann

dabei u. a. durch den Import von MOF-basierten Beschreibungen von Metamodellen im *XMI*-Format (*XML Metadata Interchange*) [OMG00b] geschehen. Wir setzen das TOY-Repository in Kombination mit einer entsprechenden Spezifikation des UML-Metamodells ein, auf das wir das CDL-API in Abschnitt 9.1.2 abgebildet haben. Eine alternative Implementierung des CDL Component Repository auf der Grundlage des *Microsoft Repository* wurde nicht weiter verfolgt, da die Entwicklung des Microsoft Repository zwischenzeitlich eingestellt wurde.

Die Implementierung des CDL Component Servers greift über das *TOY Java API* auf das TOY-Repository zu, das aus einem *TOY Server* und einem *TOY DBMS* aufgebaut ist. Das TOY Java API definiert eine kompakte Menge von Klassen, die den Zugriff auf das Repository, darin abgelegte Metadatenobjekte und deren Eigenschaftsbeschreibungen gestattet.

9.2.2 Terminologieverwaltung

Ähnlich wie der CDL Component Manager ist auch der Terminology Manager in Form einer Java-Applikation realisiert worden, die als „Thin Client“ für die Interaktion mit dem Benutzer verantwortlich ist und über RMI mit dem Terminology Server kommuniziert. Abbildung 9.11 zeigt drei Screenshots des Terminology Managers, auf denen neben der Verwaltung von Konzepten (links) auch die Auswahl von Termen (rechts oben) zur Manipulation von Sätzen (unten) dargestellt ist.

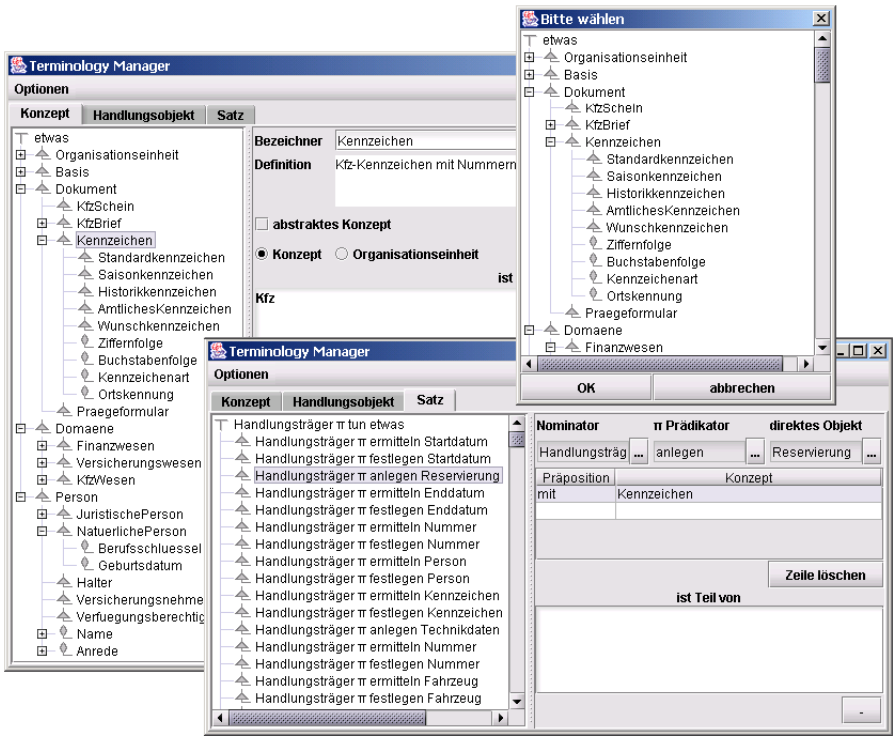


Abbildung 9.11: Terminology Manager

Der Terminology Server, der die für die Verwaltung von Terminologien erforderliche Anwendungslogik implementiert, ist auf der Grundlage der *Karlsruhe Ontology and Semantic Web Tool Suite (KAON)* [KAO03] realisiert worden. KAON ist eine quell-offene Infrastruktur für die Entwicklung von Ontologien, die neben der Modellierung konzeptueller Strukturen auch die Verwaltung von Informationen in diesen Strukturen sowie deren persistente Speicherung unterstützt. Das konzeptuelle Modell für die Repräsentation von Ontologien umfasst im Kern zwei Konstrukte: Ein *Konzept* beschreibt eine Menge von Instanzen, die spezifische, durch die Definition des Konzepts vorgegebene *Eigenschaften* aufweisen. Eine Eigenschaft muss einem Konzept zugewiesen sein. Ihr Wertebereich kann entweder auf Literale beschränkt oder durch eine nicht-leere Menge von Konzepten definiert sein. Eigenschaften, die Konzepte miteinander verbinden, können als transitiv oder symmetrisch gekennzeichnet sein. Konzepte und Eigenschaften können über Subordinationsbeziehungen hierarchisch angeordnet sein [MMV02].

Das *KAON API* stellt eine Java-Schnittstelle zur Verfügung, mit der Ontologien auf der Grundlage dieses konzeptuellen Modells verwaltet werden können. Zwecks Implementierung des Terminology-APIs aus Abbildung 9.6 auf der Grundlage des KAON APIs wurden die Klassen zur Repräsentation von Termen, Sätzen und semantischen Fällen direkt auf Konzepte abgebildet. Beziehungen zwischen den einzelnen Klassen konnten durch Eigenschaften dargestellt werden, wobei zur Repräsentation von Generalisierungs-/Spezialisierungs- und Ganzes/Teile-Beziehungen transitive Eigenschaften verwendet wurden. Die in einer Terminologie zu verwaltenden Terme, Sätze und semantischen Fälle wurden von KAON als Instanzen der Konzepte persistent abgelegt.

Für die persistente Speicherung einer Ontologie im Back-End bietet KAON zwei Varianten an. Neben dem *RDF Server*, der die Ablage gemäß dem *Resource Description Framework (RDF)* [W3C99] ermöglicht, steht der *Engineering Server* zur Verfügung, mit dem eine Ontologie (bei Bedarf vermittelt über einen J2EE-Server) in einer relationalen Datenbank abgelegt wird. Bei der Implementierung haben wir uns aufgrund des geringeren Einarbeitungs- und Konfigurationsaufwands für den Einsatz des Engineering Servers mit direkter Anbindung an eine *Oracle8i*-Datenbank entschieden.

9.2.3 Komponentensuche

Wie bereits aus dem in Abschnitt 9.1.4 vorgestellten Entwurf ersichtlich ist, haben wir die Werkzeuge zur Unterstützung der Komponentensuche als Web-Anwendung realisiert. Über das Component Retrieval Interface, das als serverseitig generierter HTML-Client in einem Web-Browser genutzt werden kann, lassen sich Suchanfragen an die Component Matching Engine stellen sowie der Bestand verfügbarer Komponentenbeschreibungen direkt durchsuchen und betrachten. Als Suchanfragen werden dabei lineare Prozessmodelle akzeptiert, die in dem in Anhang B angegebenen XML-Format spezifiziert sind. Abbildung 9.12 zeigt die graphische Präsentation einer Komponente durch das Component Retrieval Interface.

Bei der Betrachtung von Komponentenbeschreibungen besteht die Möglichkeit, detailliertere Informationen über die von der Komponente implementierten Klassen, Operationen etc. durch das Verfolgen entsprechender Links zu erhalten. Auf analoge Weise kann der Graph zur Darstellung des Komponentenverhaltens, der wie bereits beim CDL

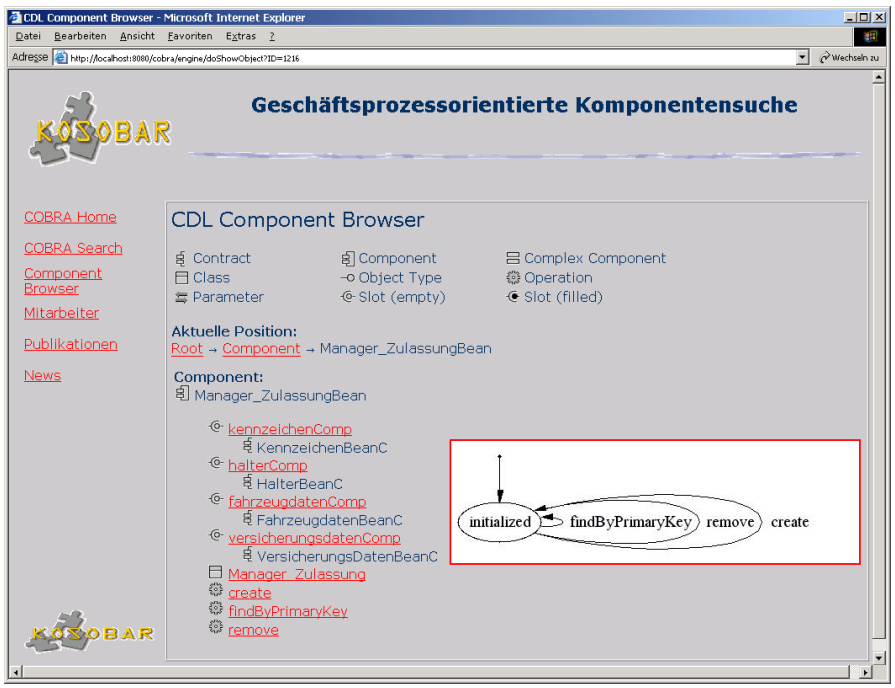


Abbildung 9.12: Component Retrieval Interface (CDL Component Browser)

Component Manager mittels der Graphenbibliothek *Grappa* erzeugt wurde, in einer vergrößerten Darstellung angezeigt werden.

Neben der Bereitstellung von Komponentenbeschreibungen im HTML-Format übernimmt das Component Retrieval Interface auch die in der Präsentations- und Evaluationsphase des Makroprozesses geschäftsprozessorientierter Komponentensuche (vgl. Abschnitt 5.2) geforderte Präsentation von Suchergebnissen. Abbildung 9.13 zeigt exemplarisch das Ergebnis einer Suchanfrage, der von der Component Matching Engine zwei Komponenten zugeordnet wurden. Zwecks Evaluation der Suchresultate durch den Verwender lässt sich die jeweilige Unterstützung des angefragten Geschäftsprozessmodells durch die gefundenen Komponenten graphisch darstellen. Dabei werden die Aktivitäten, die von der gefundenen Komponente (oder einer ihrer Klassen) direkt unterstützt werden, und solche, die von der gefundenen Komponente indirekt, d. h. über Komponenten, mit denen die Slots der gefundenen Komponente zu parametrisieren sind, durch unterschiedliche farbliche Kennzeichnungen hervorgehoben. Wie bereits bei der Darstellung des Komponentenverhaltens kommt auch hier die Graphenbibliothek *Grappa* zum Einsatz.

Grundlage der Präsentation von Suchergebnissen durch das Component Retrieval Interface sind die Informationen, die von der Component Matching Engine geliefert werden. Für die Realisierung der Component Matching Engine fand wieder die Graphenbibliothek *Grappa* Verwendung, auf deren Grundlage die Transitionssysteme zur Darstellung von Geschäftsprozessmodellen und Verhaltensbeschreibungen intern repräsentiert wurden. Daneben wurde für die Generierung der Parser zu Verarbeitung von CDL-Verhaltensbeschreibungen und -Operationsrümpfen der Parser-Generator *JavaCC* eingesetzt.

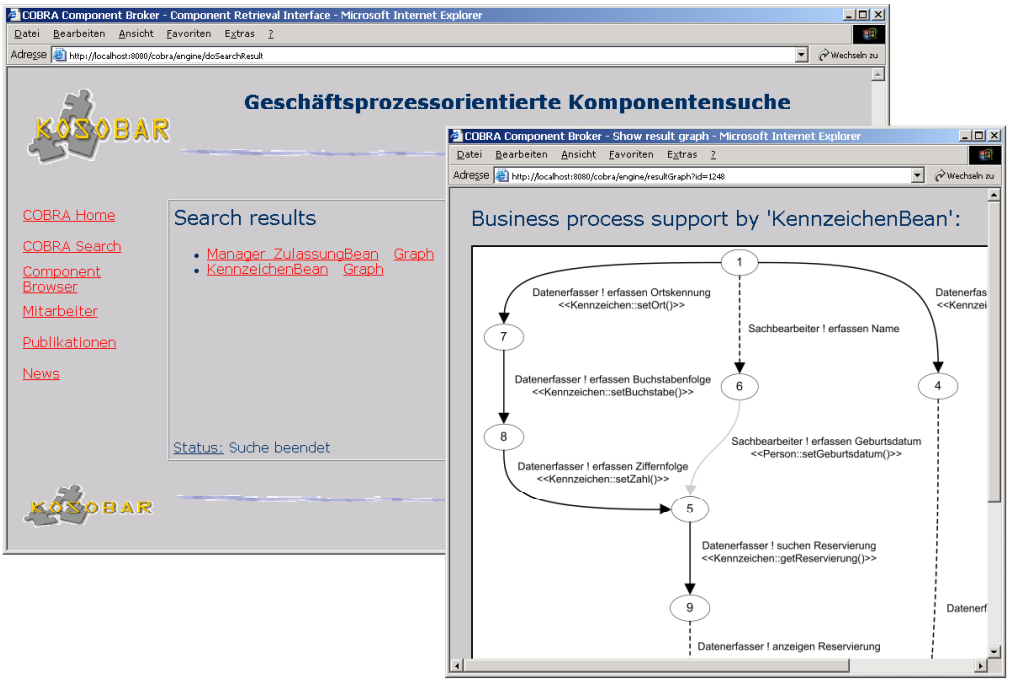


Abbildung 9.13: Component Retrieval Interface (Präsentation der Suchergebnisse)

Als Servlet-Container für das Component Retrieval Interface sowie die Component Matching Engine kam *Tomcat* aus dem Open-Source-Projekt *Apache Jakarta* in der Version 4.1 zum Einsatz.

9.3 Zusammenfassung

In diesem Kapitel haben wir ausgewählte Aspekte des Entwurfs und der Implementierung von Werkzeugen zur Verwaltung von Komponenten und Terminologien sowie zur Komponentensuche vorgestellt. Ausgehend von den zentralen Anwendungsfällen des Makroprozesses geschäftsprozessorientierter Komponentensuche haben wir zunächst eine abstrakte Systemarchitektur eingeführt, die verschiedene client- und serverseitige Werkzeuge zur Unterstützung der Aufgaben von Komponentenhersteller, Normierungsgremium und Komponentenverwender umfasst. Bei der Beschreibung des Entwurfs dieser Werkzeuge haben wir uns auf die serverseitigen Anwendungsdienste konzentriert. Im Rahmen der Dokumentation der prototypischen Implementierung haben wir schließlich die Systemarchitektur hinsichtlich der verwendeten Programmiersprache, genutzter Fremdkomponenten sowie eingesetzter Kommunikationsprotokolle konkretisiert und die realisierten Werkzeuge übersichtsartig vorgestellt.

Kapitel 10

Evaluation

Dieses Kapitel fasst die Evaluation der in Teil II dieser Arbeit entwickelten Konzepte und ihrer prototypischen Umsetzung zusammen. Dazu wird in Abschnitt 10.1 zunächst die Fallstudie vorgestellt, anhand der wir die geschäftsprozessorientierte Komponentensuche evaluiert haben. Anschließend dokumentieren, analysieren und bewerten wir in Abschnitt 10.2 die Resultate, die im Rahmen dieser Fallstudie erzielt wurden. Abschnitt 10.3 diskutiert den Entwurf und die Implementierung der KOSOBAR-Werkzeuge und nennt Verbesserungspotentiale. Die wichtigsten Erkenntnisse, die wir aus dieser Evaluation gewinnen konnten, fasst Abschnitt 10.4 abschließend zusammen.

10.1 Fallstudie: Kfz-Zulassungswesen

In diesem Abschnitt stellen wir zunächst kurz den Zweckverband *Kommunale Datenverarbeitung Oldenburg* vor, der uns für die Evaluation unserer Konzepte zur geschäftsprozessorientierten Komponentensuche Geschäftsprozessmodelle und Komponenten aus dem Anwendungsbereich des Kfz-Zulassungswesens zur Verfügung gestellt hat. Anschließend skizzieren wir den Ablauf der Fallstudie, die wir auf der Grundlage dieser Daten durchgeführt haben.

10.1.1 KDO – Kommunale Datenverarbeitung Oldenburg

Die *Kommunale Datenverarbeitung Oldenburg (KDO)* (http://www.kdo.de/) ist ein langfristiger Kooperationspartner von OFFIS, der im Jahre 1971 als Gemeinschaftseinrichtung von kommunalen Gebietskörperschaften im Raum Weser-Ems gegründet wurde. Heute ist die KDO ein von Kommunen partnerschaftlich getragenes Software- und Beratungshaus für die vielfältigen Aufgabenbereiche von Städten, Landkreisen, Gemeinden und Verbänden. Das Geschäftsgebiet reicht vom oldenburgisch-ostfriesischen Raum über die Elbe-Weser-Region bis zum Bereich des westlichen Regierungsbezirks Hannover. Darüber hinaus unterhält die KDO Kundenbeziehungen zu Kommunen in ganz Niedersachsen, Schleswig-Holstein und Hessen.

Das Dienstleistungsangebot der KDO reicht von der Planung der örtlichen technischen Infrastruktur über die fachlich-organisatorische Beratung, die Entwicklung und Bereitstellung von Software bis hin zur Einsatzunterstützung und Hilfe im Fehlerfall. Im Bereich

der Softwareentwicklung kann die KDO auf ein umfassendes Portfolio verweisen, das u. a. die kommunalen Geschäftsprozesse in den Bereichen Personalabrechnung, Finanzverwaltung, Einwohner- und Ausländerwesen sowie Kraftfahrzeugzulassungswesen unterstützt. Die Angebotspalette wird durch den Betrieb zentraler Server und eines Weitverkehrsnetzes mit der erforderlichen Sicherheitstechnologie ergänzt.

Aktuell arbeitet die KDO in einem Projekt auf dem Gebiet moderner Softwaretechnologien eng mit OFFIS zusammen. Dabei werden eine Architektur sowie ein Vorgehensmodell für die komponentenbasierte Entwicklung kommunaler Software in Java entwickelt, wobei Anforderungen des eGovernment eine wichtige Rolle einnehmen.

10.1.2 Fallstudie

Für eine umfassende Evaluation der in dieser Arbeit entwickelten Konzepte zur geschäftsprozessorientierten Komponentensuche wäre es wünschenswert gewesen, einerseits auf eine Vielfalt von Geschäftsprozessmodellen zurückgreifen zu können, mit denen reale, bei Verwendern durchgeführte Geschäftsprozesse dokumentiert werden, und andererseits einen möglichst großen Vorrat an CDL-Beschreibungen von domänenspezifischen und -unabhängigen Komponenten zur Verfügung zu haben. Da sowohl Geschäftsprozessmodelle als auch Komponentenbeschreibungen im Allgemeinen detaillierte Informationen über die internen Abläufe und Kompetenzen eines Verwenders bzw. Herstellers preisgeben, zeigen sich diese Akteure hinsichtlich der Herausgabe entsprechender Informationen sowie deren Veröffentlichung in einer wissenschaftlichen Arbeit noch zögerlich. Aus diesem Grund hat sich insbesondere der Zugang zu Komponenten über den direkten Kontakt zu deren Herstellern als problematisch erwiesen. Beschreibungen der auf aktuellen Komponentenmärkten angebotenen Komponenten genügen wie bereits in Abschnitt 4.2 dargestellt nicht den Anforderungen der geschäftsprozessorientierten Komponentensuche und eignen sich daher nicht für deren Evaluation.

Angesichts dieser Probleme bei der Datenbeschaffung haben wir im Rahmen dieser Arbeit auf eine „Minimal-Evaluation“ zurückgegriffen, bei der Geschäftsprozessmodelle und Komponenten(beschreibungen) aus derselben Hand und derselben Anwendungsdomäne zum Einsatz kamen. Die KDO hat für diesen Zweck freundlicherweise Geschäftsprozessmodelle und Komponenten aus dem Anwendungsbereich des Kfz-Zulassungswesens zur Verfügung gestellt, die im Rahmen ihrer Umstellungsbemühungen auf moderne Softwaretechnologien entwickelt wurden. Dabei handelt es sich im Detail um Aktivitätsdiagramme, mit denen die im Zulassungswesen zu unterstützenden Geschäftsprozesse modelliert wurden, und prototypisch implementierte EJB-Komponenten, die im Zuge einer J2EE-Realisierung des Kfz-Zulassungswesens entwickelt wurden. Aufgrund der allein durch den Entwicklungsprozess gegebenen Nähe dieser Geschäftsprozessmodelle und Komponenten können die im folgenden Abschnitt vorgestellten Resultate nur als erster Anhaltspunkt für Stärken und Schwächen der in dieser Arbeit vorgeschlagenen Konzepte dienen.

Das Vorgehen bei der Evaluation der geschäftsprozessorientierten Komponentensuche anhand dieser Daten lässt sich wie folgt skizzieren:

1. Erstellung einer Terminologie mit allgemeinen und domänenspezifischen Begriffen

2. Konvertierung der Aktivitätsdiagramme in lineare Prozessmodelle

3. Beschreibung der Komponenten mittels CDL
4. Geschäftsprozessorientierte Komponentensuche für ausgewählte Geschäftsprozessmodelle; dabei getrennte Untersuchung bzgl.
   (a) lokalem Alphabet (Unterstützung der Geschäftsprozesse nur durch direkte und indirekte Schnittstellen der untersuchten Komponente)
   (b) globalem Alphabet (Unterstützung der Geschäftsprozesse durch direkte und indirekte Schnittstellen der untersuchten Komponente sowie darüber referenzierte indirekte Schnittstellen genutzter Kontrakte)
5. Auswertung der Suchergebnisse und Analyse ihrer Entstehung.

Die im ersten Schritt dieses Vorgehens entstandene Terminologie wird in Abschnitt C.1 des Anhangs dieser Arbeit dokumentiert. Abschnitt C.2 fasst die im zweiten Schritt spezifizierten linearen Prozessmodelle zusammen, während Abschnitt C.3 die im dritten Schritt erstellten CDL-Beschreibungen der elf von der KDO bereitgestellten EJB-Komponenten wiedergibt. Unter diesen Komponenten befinden sich neben nur zwei Session Beans (Prozesskomponenten) ein recht hoher Anteil von neun Entity Beans (Geschäftsobjektkomponenten), die aufgrund der Eigenschaften der mit ihnen verwalteten Geschäftsobjekte allerdings oft keinen ausgeprägten Lebenszyklus aufweisen. Da die aktuelle Implementierung des semantischen Vergleichs in der Component Matching Engine keine aufbauorganisatorischen Aspekte betrachtet, haben wir bei der Beschreibung der fachlichen Semantik dieser Komponenten auf die aufbauorganisatorische Zuordnung der Operationen im Sinne eines Berechtigungskonzepts (vgl. Abschnitt 8.3.1) verzichtet.

Hinsichtlich der auf der Grundlage dieser Beschreibungen durchgeführten geschäftsprozessorientierten Komponentensuche haben wir zwei Varianten untersucht. Zunächst wurde beim semantischen Vergleich von Geschäftsprozessmodell und Komponente nur das *lokale* Alphabet der Komponente, d. h. nur die Operationen ihrer direkten und indirekten Schnittstellen betrachtet. Die Komponenten, die bzgl. des lokalen Alphabets eine semantische Unterstützung des Geschäftsprozessmodells bieten, haben wir anschließend noch einmal hinsichtlich des *globalen* Alphabets untersucht. Dabei wurden zusätzlich die Schnittstellen der über die transitive Hülle der Verweisrelation erreichbaren Geschäftsobjekte in den semantischen Vergleich einbezogen. Die Wahl des Alphabets hatte Einfluss auf die Menge der beim anschließenden Protokollvergleich in Geschäftsprozessmodell und Komponente versteckten Aktionen.

10.2 Bewertung der geschäftsprozessorientierten Komponentensuche

In diesem Abschnitt stellen wir die im Rahmen der Fallstudie ermittelten Suchergebnisse vor, auf deren Grundlage wir Stärken und Schwächen der geschäftsprozessorientierten Komponentensuche analysieren. Dazu gehen wir zunächst einzeln auf die bzgl. der verfügbaren Geschäftsprozessmodelle erzielten Resultate ein, bevor wir eine zusammenfassende Gesamtbewertung vornehmen.

10.2.1 Bewertung individueller Suchergebnisse

Für die Dokumentation der Suchergebnisse verwenden wir im Folgenden ein einheitliches Schema, bei dem die Unterstützung des jeweils betrachteten Geschäftsprozessmodells in tabellarischer Form festgehalten ist. Eine solche Tabelle enthält Spalten für die Namen der untersuchten Komponenten, ihre intuitiv bewertete Relevanz für die Unterstützung der durch das Geschäftsprozessmodell spezifizierten Abläufe sowie die Ergebnisse der geschäftsprozessorientierten Komponentensuche auf der Semantik- und der Protokollebene. Letztere Ergebnisse führen wir sowohl für die Betrachtung des lokalen als auch des globalen Alphabets der Komponente auf.

Nachfolgend besprechen wir die Resultate der geschäftsprozessorientierten Komponentensuche geordnet nach den untersuchten Geschäftsprozessmodellen. Dabei konzentrieren wir uns auf die Erläuterung von unerwarteten Ergebnissen sowie von Auffälligkeiten, die bei der Entstehung einzelner Suchresultate zu beobachten waren. Für die quantitative Bewertung der Resultate durch Kennzahlen für Präzision und Trefferquote ist es erforderlich, die Relevanz einer Komponente für die Unterstützung der durch ein Geschäftsprozessmodell spezifizierten Prozesse intuitiv bewerten zu können. Zu diesem Zweck betrachten wir in der folgenden Evaluation lediglich die Geschäftsprozessmodelle „Kennzeichen auswählen“, „Halterdaten erfassen“, „Fahrzeugdaten erfassen“, „Versicherungsdaten erfassen“ und „Händlerzulassung einpflegen“, da diesen geeignete (und damit relevante) Komponenten auf einfache und nachvollziehbare Weise eindeutig zugeordnet werden können. Wir verzichten dagegen auf die explizite Berücksichtigung der Geschäftsprozessmodelle „Halterdaten übernehmen“, „Personendaten erfassen“ und „Zulassung abschließen“, da diese von den oben genannten Prozessmodellen über Hierarchisierungsbeziehungen referenziert werden, sowie die Betrachtung der Geschäftsprozessmodelle „Neuzulassung durchführen“ und „Grunddaten erfassen“, für die aufgrund ihres hierarchischen Aufbaus nahezu alle verfügbaren Komponenten relevant erscheinen (vgl. hierzu auch Abbildung C.4 in Abschnitt C.2).

Kennzeichen auswählen Die bzgl. des Geschäftsprozessmodells „Kennzeichen auswählen“ erzielten Suchresultate sind in Tabelle 10.1 zusammengefasst. Dabei bewerten wir die intuitive Relevanz einer Komponente auf einer Skala von „sehr relevant“ (+) über „mäßig relevant“ (∘) bis „nicht relevant“ (−). Der in den mit „Semantik“ beschrifteten Spalten eingetragene Prozentwert gibt den Anteil der Aktivitäten des Geschäftsprozessmodells an, für die im Rahmen des Semantikvergleichs eine Implementierung durch eine Operation aus dem Alphabet der Komponente gefunden wurde.

Bei Betrachtung des lokalen Alphabets wurden durch die geschäftsprozessorientierte Komponentensuche fünf Komponenten gefunden, von denen wir vier als relevant eingestuft haben. Damit ergibt sich eine Präzision von $4/5 = 0,80$. Da alle relevanten Komponenten gefunden wurden, erhalten wir zudem eine hohe Trefferquote von $4/4 = 1,00$.

Bzgl. der Komponente `KennzeichenBean`, die mit 45% die höchste semantische Abdeckung des Geschäftsprozessmodells bietet, ist anzumerken, dass zwecks Unterstützung der Aktivität „Sachbearbeiter ! erfassen Kennzeichenart“ zunächst durch Ausführung der versteckten Operation `KennzeichenBean::create()` ein Kennzeichen-Geschäftsobjekt erzeugt wurde. Die anschließende Suche nach einem Wunschkennzeichen, die durch

Komponente	**Relevanz**	**Suchergebnis**			
		lokal		**global**	
		Semantik	**Protokoll**	**Semantik**	**Protokoll**
AdresseBean	–	0 %	ignoriert	ignoriert	ignoriert
Dienst_-KennzeichenBean	∘	9 %	√	54 %	–
FahrzeugdatenBean	–	0 %	ignoriert	ignoriert	ignoriert
HalterBean	–	0 %	ignoriert	ignoriert	ignoriert
KennzeichenBean	+	45 %	√	54 %	√
KfzBean	–	0 %	ignoriert	ignoriert	ignoriert
Manager_-ZulassungBean	∘	9 %	√	63 %	–
OFDDatenBean	–	0 %	ignoriert	ignoriert	ignoriert
PersonBean	–	9 %	√	45 %	√
ReservierungBean	+	18 %	√	63 %	√
Versicherungsdaten-Bean	–	0 %	ignoriert	ignoriert	ignoriert

Tabelle 10.1: Suchresultate für Geschäftsprozessmodell „Kennzeichen auswählen“

die Operation `KennzeichenBean::findByPrimaryKey()` realisiert wird, referenziert ein zweites Kennzeichen. Dieses unerwünschte Verhalten ist durch die Unterspezifikation des Geschäftsprozessmodells hinsichtlich des Datenflusses und die Beschränkung des Protokollvergleichs auf die Betrachtung der Ablauforganisation zu begründen. Die Komponenten `Dienst_KennzeichenBean` und `Manager_ZulassungBean` sind für geeignet befunden worden, weil sie Unterstützung bei der Suche nach einem Wunschkennzeichen versprechen.

Die Komponente `PersonBean`, die wir als nicht relevant eingestuft haben, ist gefunden worden, weil sie die Ausführung der Aktivität „Sachbearbeiter ! erfassen Geburtsdatum“ durch die Operation `Person::setGeburtsdatum()` unterstützt. Diese semantische Zuordnung erscheint durchaus sinnvoll, obgleich mit dem untersuchten Geschäftsprozessmodell nicht die durch diese Komponente implementierte Verwaltung von Personendaten modelliert wird. Bemerkenswert ist darüber hinaus, dass der semantische Vergleich keine Unterstützung für die Aktivität „Sachbearbeiter ! erfassen Name“ finden konnte. Dies ist u. a. damit zu begründen, dass die Komponente `PersonBean` Operationen zur getrennten Verwaltung von Vor- und Nachnamen implementiert. Zwar sind „Vorname“ und „Nachname“ in der Terminologie als Spezialisierung von „Name“ definiert, allerdings stellt das in der Komponentenbeschreibung verwendete Prädikat „festlegen“ eine Generalisierung des geforderten „erfassen“ dar. Aufgrund dieser gegenläufigen Generalisierungs-/Spezialisierungsbeziehungen konnte durch den semantischen Vergleich keine derartige Beziehung

zwischen den normsprachlichen Sätzen festgestellt werden. Eine ergänzende Berücksichtigung von Ganzes/Teile-Beziehungen (wie hier zwischen dem Ganzen „Name“ und seinen Teilen „Vorname“ und „Nachname“) beim Semantikvergleich könnte in diesen Situationen hilfreich sein.

Bei Betrachtung der Suchresultate unter Verwendung des globalen Alphabets ist zunächst festzustellen, dass nur noch drei Komponenten für geeignet befunden worden sind. Während die Präzision nun lediglich einen Wert von $2/3 = 0,66$ erreicht, sinkt die Trefferquote sogar auf niedrige $2/4 = 0,50$ ab. Aus der Menge der bei Betrachtung des lokalen Alphabets gefundenen Komponenten bleiben nunmehr nur noch die Komponente `KennzeichenBean`, die nach intuitiver Einschätzung auch die „beste“ Komponente darstellt, sowie die Komponenten `ReservierungBean` und `PersonBean` übrig. Die geschäftsprozessorientierte Komponentensuche stellt bei diesen Komponenten nun die Möglichkeit der Unterstützung weiterer Aktivitäten zur Verwaltung von Personen- und Kennzeichendaten fest. Dabei ist allerdings anzumerken, dass das dafür akzeptierte Maß an Process Interference die spezifizierten Geschäftsprozesse stark verfremdet.

Die Tatsache, dass nunmehr die Komponenten `Dienst_KennzeichenBean` und `Manager_ZulassungBean` nicht mehr für geeignet befunden worden sind, ist damit zu erklären, dass der Protokollvergleich bei Berücksichtigung des globalen Alphabets im Allgemeinen eine stärkere Filterwirkung entfaltet. Da – wie aus Tabelle 10.1 ersichtlich ist – der semantische Vergleich eine deutlich höhere Abdeckung des Geschäftsprozessmodells durch die einzelnen Komponenten ermittelt, werden beim Protokollvergleich weniger Aktionen versteckt und damit höhere Anforderungen an die Kompatibilität der Protokolle gestellt. Im konkreten Fall der Komponenten `Dienst_KennzeichenBean` und `Manager_ZulassungBean` wurde die Aktivität „Datenerfasser ! erfassen Kennzeichenart“ nun der Operation `Kennzeichen::setType()` zugeordnet und daher nicht mehr versteckt. Da die Operationen `Dienst_Kennzeichen::holeNaechstesFreiesAKZ()` und `Dienst_Kennzeichen::sucheNaechstesKennzeichen()` der Aktivität „Datenerfasser ! suchen Wunschkennzeichen“ zugeordnet und daher ebenfalls nicht versteckt wurden, war es beim Protokollvergleich nicht möglich, eine für die geforderte Bearbeitung von Kennzeicheneigenschaften benötigte Referenz auf ein Kennzeichen-Geschäftsobjekt zu erhalten, ohne dass zuvor explizit nach einem Wunschkennzeichen gesucht wurde.

Insgesamt lässt sich bzgl. der Betrachtung des globalen Alphabets beim semantischen Vergleich festhalten, dass die bewirkte erhöhte Filterwirkung höhere Ansprüche an „korrekt“ modellierte Geschäftsprozesse stellt. Bevor ein Geschäftsobjekt im Verlauf eines Geschäftsprozesses manipuliert werden kann, sollte in irgendeiner Form eine Referenz auf dieses Objekt deklariert werden (z. B. durch Aktivitäten zur Erzeugung oder Suche von Geschäftsobjekten). Dies kann durch Berücksichtigung des Datenflusses bei der Geschäftsprozessmodellierung erreicht werden, wobei dieser allerdings auf der Instanz- und nicht (wie z. B. bei EPKs üblich) auf der Typebene zu spezifizieren ist.

Halterdaten erfassen Die Ergebnisse der geschäftsprozessorientierten Komponentensuche auf der Grundlage des Geschäftsprozessmodells „Halterdaten erfassen“ sind in Tabelle 10.2 zusammengefasst. Wir bewerten vier Komponenten als grundsätzlich relevant für die Unterstützung der mit diesem Geschäftsprozessmodell spezifizierten Abläufe.

Komponente	**Relevanz**	**Suchergebnis**			
		lokal		**global**	
		Semantik	**Protokoll**	**Semantik**	**Protokoll**
AdresseBean	∘	0 %	ignoriert	ignoriert	ignoriert
Dienst_-KennzeichenBean	–	0 %	ignoriert	ignoriert	ignoriert
FahrzeugdatenBean	–	0 %	ignoriert	ignoriert	ignoriert
HalterBean	+	21 %	√	63 %	√
KennzeichenBean	–	15 %	√	57 %	√
KfzBean	–	0 %	ignoriert	ignoriert	ignoriert
Manager_-ZulassungBean	+	31 %	√	78 %	√
OFDDatenBean	–	0 %	ignoriert	ignoriert	ignoriert
PersonBean	+	26 %	√	57 %	√
ReservierungBean	–	0 %	ignoriert	ignoriert	ignoriert
Versicherungsdaten-Bean	–	5 %	√	57 %	√

Tabelle 10.2: Suchresultate für Geschäftsprozessmodell „Halterdaten erfassen“

Die geschäftsprozessorientierte Komponentensuche hat unter Berücksichtigung des lokalen Alphabets fünf Komponenten gefunden, von denen jedoch lediglich drei relevant erscheinen. Wir erhalten daher für die Komponentensuche bzgl. des Geschäftsprozessmodells „Halterdaten erfassen“ eine Präzision von $3/5 = 0,60$ sowie eine Trefferquote von $3/4 = 0,75$.

Bemerkenswert an dem erzielten Suchresultat ist zunächst einmal die Tatsache, dass die Komponente `AdresseBean`, die wir aufgrund der Aktivität „Datenerfasser ! erfassen Adresse“ als relevant bewertet haben, nicht gefunden wurde. Dies ist damit zu begründen, dass diese Komponente Operationen zur Verwaltung von „Teilen“ des „Ganzen“ „Adresse“ anbietet (z. B. Straße, Hausnummer, Postleitzahl und Ort), wir aber Ganzes/Teile-Beziehungen in der aktuellen Form der geschäftsprozessorientierten Komponentensuche nicht berücksichtigen. Die Relevanz der `AdresseBean` oder einer vergleichbaren Komponente wird allerdings dadurch unterstrichen, dass die gefundene Komponente `PersonBean` einen Erweiterungspunkt für Komponenten zur Adressverwaltung definiert.

Die von uns als nicht relevant eingestufte Komponente `KennzeichenBean` ist aufgrund der Spezifizierung von Aktivitäten zur Eingabe von Kennzeichendaten gefunden worden. Da diese Daten im Rahmen des Geschäftsprozesses „Halterdaten erfassen“ nicht persistent verwaltet werden sollen, sondern lediglich der Suche nach Halterdaten dienen, die zu einem früheren Zeitpunkt angelegt wurden, wird diese Komponente tatsächlich nicht benötigt. Die geschäftsprozessorientierte Komponentensuche hat allerdings keine

Möglichkeit, zwischen der reinen Eingabe von Daten und deren Erfassung im Sinne einer persistenten Speicherung zu unterscheiden, da in der verwendeten Terminologie eine solche begriffliche Differenzierung nicht vorgesehen ist.

Wie erwartet wurden die Komponenten `HalterBean` und `PersonBean` gefunden, mit denen personenbezogene Daten zu Haltern von Kraftfahrzeugen verwaltet werden können. Die Komponente `Manager_ZulassungBean` unterstützt die modellierten Geschäftsprozesse durch Operationen zur Erfassung sowie zur anschließenden Speicherung von Halterdaten. Dabei ist allerdings einschränkend anzumerken, dass die Zuordnung der verfügbaren Operationen zu den geforderten Aktivitäten nicht immer erwartungsgemäß erfolgt ist. Die `VersicherungsdatenBean` wurde entgegen unseren Erwartungen als geeignete Komponente identifiziert, da die von ihr unterstützte Aktivität „Datenerfasser ! bearbeiten Hinweis“ aus terminologischer Sicht unterspezifiziert ist.

Die Berücksichtigung des globalen Alphabets bei der Komponentensuche führt zu Resultaten, die von denen abweichen, die wir bzgl. des Geschäftsprozessmodells „Kennzeichen auswählen“ beobachten konnten. Die Anzahl der gefundenen Komponenten und damit auch die Werte für Präzision und Trefferquote bleiben konstant, d. h. die Verwendung des globalen Alphabets hat die Filterwirkung des Protokollvergleichs bei Betrachtung des Geschäftsprozessmodells „Halterdaten erfassen“ nicht verstärkt.

Die Komponente `HalterBean` bietet nunmehr neben den Operationen zur Manipulation von Halterdaten indirekt auch Unterstützung für diejenigen Aktivitäten an, die sich auf die Eingabe von Kennzeichendaten und die Verwaltung von Personen im Allgemeinen beziehen. Dies wird durch die Existenz von Operationen wie z. B. `getPerson()` ermöglicht, die den Zugriff auf Geschäftsobjekte von Komponenten gestatten, mit denen die Erweiterungspunkte der `HalterBean` parametrisiert werden müssen. Da Referenzen auf diese Geschäftsobjekte aufgrund der spezifizierten Protokolle erst erlangt werden können, nachdem diese über schreibende Zugriffe definiert wurden, ist das Geschäftsprozessmodell „Halterdaten erfassen“ durch den Protokollvergleich im Sinne der Component Interference um entsprechende Aktivitäten erweitert worden. Diese Erweiterungen sind durchaus als sinnvoll einzuschätzen, setzen jedoch eine Möglichkeit zur Erzeugung von Geschäftsobjekten zur Repräsentation von Kennzeichen- und Personendaten voraus, wie sie z. B. die Komponenten `KennzeichenBean` und `PersonBean` bieten. Eine explizite Modellierung des Datenflusses in Geschäftsprozessmodellen sowie deren Berücksichtigung bei der geschäftsprozessorientierten Komponentensuche würde auch hier zu klareren Ergebnissen führen.

Die Komponente `Manager_ZulassungBean` bietet bei Berücksichtigung des globalen Alphabets ein auffällig hohes Maß an semantischer Abdeckung an. Dabei werden neben der bereits bzgl. des lokalen Alphabets festgestellten Unterstützung bei der Erfassung und Speicherung von Halterdaten nun auch weitere Aktivitäten mit Bezug zur Verwaltung von Personen- und Kennzeichendaten unterstützt. Diese hohe semantische Abdeckung wird allerdings nur durch Betrachtung langer Verweisketten zwischen Geschäftsobjekten ermöglicht, so dass die tatsächliche Eignung der Komponente für die Unterstützung der modellierten Geschäftsprozesse verzerrt wiedergegeben wird. Ähnliche Verweisketten treten auch bei der durch die `PersonBean` zusätzlich angebotenen Unterstützung für die Verwaltung von Kennzeichendaten auf. Die Tatsache, dass diese Komponente nun neben der Bearbeitung von allgemeinen personenbezogenen Daten auch indirekt den Zugriff auf Halterdaten ermöglicht, erscheint allerdings durchaus sinnvoll.

Fahrzeugdaten erfassen Tabelle 10.3 fasst die Resultate der geschäftsprozessorientierten Komponentensuche zusammen, die bzgl. des Geschäftsprozessmodells „Fahrzeugdaten erfassen“ erzielt wurden. Wir bewerten hier fünf Komponenten als grundsätzlich relevant, wobei die Komponenten `FahrzeugdatenBean`, `Manager_ZulassungBean` und `PersonBean` besonders wichtig erscheinen.

Komponente	**Relevanz**	**Suchergebnis**			
		lokal		**global**	
		Semantik	**Protokoll**	**Semantik**	**Protokoll**
AdresseBean	∘	0 %	ignoriert	ignoriert	ignoriert
Dienst_-KennzeichenBean	–	0 %	ignoriert	ignoriert	ignoriert
FahrzeugdatenBean	+	61 %	√	69 %	√
HalterBean	–	0 %	ignoriert	ignoriert	ignoriert
KennzeichenBean	–	0 %	ignoriert	ignoriert	ignoriert
KfzBean	–	0 %	ignoriert	ignoriert	ignoriert
Manager_-ZulassungBean	+	65 %	√	80 %	√
OFDDatenBean	∘	7 %	√	69 %	√
PersonBean	+	15 %	√	69 %	√
ReservierungBean	–	0 %	ignoriert	ignoriert	ignoriert
Versicherungsdaten-Bean	–	0 %	ignoriert	ignoriert	ignoriert

Tabelle 10.3: Suchresultate für Geschäftsprozessmodell „Fahrzeugdaten erfassen“

Die bei Berücksichtigung des lokalen Alphabets ermittelten Resultate decken sich weitgehend mit unseren Erwartungen. Die geschäftsprozessorientierte Komponentensuche hat vier der elf verfügbaren Komponenten als geeignet bewertet. Da wir diese Komponenten allesamt als relevant eingestuft haben, erhalten wir bzgl. der Präzision den Maximalwert von $4/4 = 1,00$ und erreichen eine Trefferquote von $4/5 = 0,80$.

Wie bereits aufgrund des Namens zu erwarten war, trägt die Komponente `FahrzeugdatenBean` in großem Umfang zur Unterstützung der spezifizierten Geschäftsprozesse bei. Als einzige Einschränkung ist in diesem Zusammenhang zu erwähnen, dass die Operation `Fahrzeugdaten::setTyp()` sowohl zu der Aktivität „Datenerfasser ! erfassen Typ“ als auch zu der Aktivität „Datenerfasser ! erfassen Ausfuehrung“ zugeordnet wurde. Zur Unterstützung der letztgenannten Aktivität wäre die Operation `Fahrzeugdaten::setAusfuehrung()` besser geeignet gewesen. In der aktuellen Realisierung wird im Semantikvergleich allerdings die „erstbeste“ Operation gewählt. Die Qualität der Zuordnung von Operationen zu Aktivitäten könnte durch die Einführung einer Metrik für den Abstand zwischen Operationen und Aktivitäten bzw. den ihrer fachlichen Semantik

zugrunde liegenden normsprachlichen Sätzen gesteigert werden. Das gleiche Problem findet sich in den Suchergebnissen der Komponenten `PersonBean` und `OFDDatenBean` wieder. Hinsichtlich letztgenannter Komponente ist zu erwähnen, dass die Aktivität „Datenerfasser ! erfassen OFDDaten“ nicht erfolgreich zugeordnet werden konnte, da die Komponente keine Operationen auf einem *ValueObject* [Bie02], d. h. einem Container-Objekt mit Daten für die Oberfinanzdirektion definiert.

Neben der `FahrzeugdatenBean` bietet auch die Komponente `Manager_ZulassungBean` ein hohes Maß an Unterstützung für die modellierten Geschäftsprozesse an. Diese Unterstützung rührt daher, dass die Operation `erfasseFahrzeugdaten()` aufgrund der ihr zugeordneten fachlichen Semantik für einen großen Anteil der auszuführenden Aktivitäten eine geeignete Implementierung darstellt. Die Komponente `AdresseBean` wurde aus Gründen, die wir bereits bei der Bewertung der bzgl. des Geschäftsprozessmodells „Halterdaten erfassen“ erzielten Suchergebnisse erläutert haben, nicht gefunden.

Wie schon beim vorangegangenen Geschäftsprozessmodell ist auch hinsichtlich des Prozessmodells „Fahrzeugdaten erfassen“ keine erhöhte Filterwirkung durch die Berücksichtigung des globalen Alphabets zu beobachten. Da die Präzision allerdings bereits beim Maximalwert von $4/4 = 1,00$ gelegen hat, wäre eine solche zusätzliche Filterung auch nicht erwünscht gewesen. Insgesamt lässt sich festhalten, dass alle gefundenen Komponenten eine extrem hohe semantische Überdeckung von 69% bzw. im Falle der `Manager_ZulassungBean` sogar 80% erzielen, da sie über lange und nicht sinnvolle Verweisketten im Wesentlichen dieselbe Menge von Objekttypen (bzw. deren Instanzen) referenzieren. Die `Manager_ZulassungBean` erreicht dabei das höchste Maß semantischer Abdeckung, da sie als einzige Komponente die Aktivitäten zum Speichern von Fahrzeugdaten durch geeignete Operationen unterstützt.

Versicherungsdaten erfassen Die Resultate der geschäftsprozessorientierten Komponentensuche bzgl. des Geschäftsprozessmodells „Versicherungsdaten erfassen“ haben wir in Tabelle 10.4 zusammengestellt. Neben der Komponente `VersicherungsdatenBean` erachten wir mit `Manager_ZulassungBean`, `PersonBean` und `AdresseBean` drei weitere Komponenten als relevant für die Unterstützung der spezifizierten Geschäftsprozesse.

Die bei Betrachtung des lokalen Alphabets erzielten Ergebnisse weichen deutlich von unseren Erwartungen ab. Zwar wurden immerhin drei der vier intuitiv relevanten Komponenten gefunden, darüber hinaus sind allerdings noch drei weitere Komponenten als geeignet identifiziert worden. Damit ergibt sich ein niedriger Wert von $3/6 = 0,50$ für die Präzision bei einer durchaus akzeptablen Trefferquote von $3/4 = 0,75$.

Bei eingehenderer Betrachtung der Suchergebnisse ist zunächst einmal positiv festzuhalten, dass die Komponente `VersicherungsdatenBean` wie erwartet ein hohes Maß an semantischer Abdeckung bietet und durch die geschäftsprozessorientierte Komponentensuche gefunden wurde. Die Komponente `Manager_ZulassungBean`, die neben der Erfassung von Versicherungsdaten auch das abschließende Speichern des Erfassungsvorgangs unterstützt, wurde erwartungsgemäß ebenfalls als geeignet bewertet.

Die Komponente `HalterBean` ist fälschlicherweise gefunden worden, da die Operation `Halter::setHinweisUeberlangerName()` der Aktivität „Datenerfasser ! erfassen Vermerk“ als mögliche Implementierung zugeordnet wurde. Diese unerwünschte semantische Zuordnung lässt sich damit begründen, dass die fachliche Semantik der Opera-

Komponente	**Relevanz**	**Suchergebnis**			
		lokal		**global**	
		Semantik	**Protokoll**	**Semantik**	**Protokoll**
AdresseBean	∘	0 %	ignoriert	ignoriert	ignoriert
Dienst_-KennzeichenBean	–	0 %	ignoriert	ignoriert	ignoriert
FahrzeugdatenBean	–	0 %	ignoriert	ignoriert	ignoriert
HalterBean	–	5 %	√	70 %	√
KennzeichenBean	–	11 %	√	70 %	√
KfzBean	–	0 %	ignoriert	ignoriert	ignoriert
Manager_-ZulassungBean	+	58 %	√	88 %	√
OFDDatenBean	–	0 %	ignoriert	ignoriert	ignoriert
PersonBean	+	11 %	√	70 %	√
ReservierungBean	–	11 %	√	70 %	√
Versicherungsdaten-Bean	+	52 %	√	70 %	√

Tabelle 10.4: Suchresultate für Geschäftsprozessmodell „Versicherungsdaten erfassen“

tion bzw. der Aktivität aus terminologischer Sicht unterspezifiziert ist. Die Komponenten `KennzeichenBean` und `ReservierungBean` unterstützen die Eingrenzung eines Zeitraums durch Angabe von Start- und Enddatum, wie sie auch für die Erfassung eines Versicherungszeitraums erforderlich ist, und sind daher für geeignet befunden worden. Die `PersonBean` implementiert korrekt zugeordnete Unterstützung für die Speicherung personenbezogener Daten. Wie bereits im Zusammenhang mit dem Geschäftsprozessmodell „Kennzeichen auswählen“ erläutert wurde, konnte dabei die Aktivität „Datenerfasser ! erfassen Name“ keiner Operation der `PersonBean` zugeordnet werden. Die Gründe, aus denen die Komponente `AdresseBean` bei dem hier betrachteten Geschäftsprozessmodell nicht gefunden wurde, haben wir bereits im Zusammenhang mit dem Geschäftsprozessmodell „Halterdaten erfassen“ diskutiert.

Bei Betrachtung des globalen Alphabets ist wiederum keine erhöhte Filterwirkung des Protokollvergleichs festzustellen. Wir erhalten daher die gleichen Werte für Präzision und Trefferquote. Während die Funktionalität der Komponente `VersicherungsdatenBean` über einen Verweis nunmehr sinnvoll durch Dienste zur Verwaltung personenbezogener Daten ergänzt wird und die Komponenten `PersonBean` und `HalterBean` auf vergleichbare Weise die Verwaltung von Versicherungsdaten unterstützen, gilt für alle anderen Komponenten, dass die zusätzliche semantische Unterstützung nur über lange und ungeeignete Verweisketten erzielt wird, die keine sinnvolle Ergänzung des betrachteten Geschäftsprozessmodells im Sinne der Component Interference darstellen.

Händlerzulassung einpflegen Das Geschäftsprozessmodell „Händlerzulassung einpflegen“ unterscheidet sich von den in den vorangegangenen Abschnitten untersuchten Geschäftsprozessmodellen insbesondere dadurch, dass es einen höheren Anteil an Interaktionen zwischen einem Sachbearbeiter in einer Zulassungsstelle und einem Schalterkunden aufweist und daher in deutlich geringerem Maße Datenerfassungsvorgänge beschreibt. Da dieses Geschäftsprozessmodell auf einer recht abstrakten Ebene formuliert ist, sehen wir lediglich die Prozesskomponente `Manager_ZulassungBean` als relevant für die Unterstützung der beschriebenen Abläufe an. Die Ergebnisse der geschäftsprozessorientierten Komponentensuche bzgl. dieses Geschäftsprozessmodells haben wir in Tabelle 10.5 zusammengefasst.

Komponente	**Relevanz**	**Suchergebnis**			
		lokal		**global**	
		Semantik	**Protokoll**	**Semantik**	**Protokoll**
AdresseBean	–	0 %	ignoriert	ignoriert	ignoriert
Dienst_-KennzeichenBean	–	0 %	ignoriert	ignoriert	ignoriert
FahrzeugdatenBean	–	0 %	ignoriert	ignoriert	ignoriert
HalterBean	–	0 %	ignoriert	ignoriert	ignoriert
KennzeichenBean	–	0 %	ignoriert	ignoriert	ignoriert
KfzBean	–	0 %	ignoriert	ignoriert	ignoriert
Manager_-ZulassungBean	+	10 %	√	10 %	√
OFDDatenBean	–	0 %	ignoriert	ignoriert	ignoriert
PersonBean	–	0 %	ignoriert	ignoriert	ignoriert
ReservierungBean	–	0 %	ignoriert	ignoriert	ignoriert
Versicherungsdaten-Bean	–	0 %	ignoriert	ignoriert	ignoriert

Tabelle 10.5: Suchresultate für Geschäftsprozessmodell „Händlerzulassung einpflegen“

Die geschäftsprozessorientierte Komponentensuche hat bei Berücksichtigung des lokalen Alphabets lediglich die Komponente `Manager_ZulassungBean` gefunden, so dass wir eine Präzision und Trefferquote von $1/1 = 1,00$ erhalten. Dabei ist allerdings einschränkend festzuhalten, dass selbst für diese Komponente ein nur geringes Maß an semantischer Unterstützung ermittelt werden konnte. So deckt die `Manager_ZulassungBean` lediglich die Aktivität „Sachbearbeiter ! speichern Zulassung“ durch eine geeignete Operation ab. Die Aktivität „Sachbearbeiter ! aendern Zulassung“ findet keine semantische Unterstützung, da ihr die dafür sinnvoll einsetzbaren Operationen `erfasseHalterdaten()`, `erfasseFahrzeugdaten()` und `erfasseVersicherungsdaten()` nicht zugeordnet werden konnten. Um diese Zuordnung treffen zu können, wäre es erforderlich gewesen, einen

semantischen Zusammenhang zwischen dem Begriff „Zulassung“ einerseits und den Begriffen „Halterdaten“, „Fahrzeugdaten“ und „Versicherungsdaten“ andererseits (z. B. über eine Ganzes/Teile-Beziehung) herstellen und beim Semantikvergleich nutzen zu können. Darüber hinaus bietet die Komponente `Manager_ZulassungBean` keine Operationen an, mit denen Zulassungsvorgänge gesucht und abgebrochen, Zulassungsgebühren berechnet oder Zulassungsdokumente gedruckt werden können. Letztere Beschränkungen der Funktionalität sind mit dem prototypischen Charakter der gesamten Implementierung der Komponenten aus dem Kfz-Zulassungswesen zu begründen.

Die Betrachtung des globalen Alphabets hat keine Veränderungen bei der Komponentensuche erbracht. Da die Komponente `Manager_ZulassungBean` auch in diesem Fall als geeignet identifiziert wurde, erhalten wir die gleichen Werte für Präzision und Trefferquote.

10.2.2 Gesamtbewertung

Obgleich uns für die Evaluation nur eine geringe Anzahl von Geschäftsprozessmodellen und eine vergleichbar kleine Menge domänenspezifischer Komponenten zur Verfügung gestanden haben, wollen wir in diesem Abschnitt versuchen, eine erste zusammenfassende Bewertung der geschäftsprozessorientierten Komponentensuche vorzunehmen und Verbesserungspotentiale aufzudecken.

Die im Rahmen dieser Evaluation ermittelten Gesamtwerte für Präzision und Trefferquote der geschäftsprozessorientierten Komponentensuche ergeben sich bei Berücksichtigung des lokalen Alphabets wie folgt:

$$\begin{aligned} \textit{Präzision(lokal)} &= \frac{1}{5} \cdot \left(\frac{4}{5} + \frac{3}{5} + \frac{4}{4} + \frac{3}{6} + \frac{1}{1} \right) = 0,78 \\ \textit{Trefferquote(lokal)} &= \frac{1}{5} \cdot \left(\frac{4}{4} + \frac{3}{4} + \frac{4}{5} + \frac{3}{4} + \frac{1}{1} \right) = 0,86 \end{aligned}$$

Auf den ersten, oberflächlichen Blick sind diese Werte recht zufriedenstellend. Dieser Eindruck relativiert sich jedoch, wenn man berücksichtigt, dass die im Rahmen der Evaluation genutzten Geschäftsprozessmodelle und Komponenten bereits aufgrund ihrer Herkunft eine hohe Verwandtschaft aufweisen. Für die geschäftsprozessorientierte Komponentensuche unter Berücksichtigung des globalen Alphabets konnten die folgenden Werte für Präzision und Trefferquote ermittelt werden:

$$\begin{aligned} \textit{Präzision(global)} &= \frac{1}{5} \cdot \left(\frac{2}{3} + \frac{3}{5} + \frac{4}{4} + \frac{3}{6} + \frac{1}{1} \right) = 0,75 \\ \textit{Trefferquote(global)} &= \frac{1}{5} \cdot \left(\frac{2}{4} + \frac{3}{4} + \frac{4}{5} + \frac{3}{4} + \frac{1}{1} \right) = 0,76 \end{aligned}$$

Während sich die Präzision bei Verwendung des globalen Alphabets nur marginal gegenüber dem bei Betrachtung des lokalen Alphabets gemessenen Wert verschlechtert hat, ist die Trefferquote erkennbar gesunken. Abseits dieser quantitativen Betrachtungen muss zudem angemerkt werden, dass die auf der Grundlage des globalen Alphabets erzielten Suchergebnisse im Allgemeinen eine schlechtere Qualität aufwiesen. So waren zum

einen vielfach ungeeignete semantische Zuordnungen von Operationen zu Aktivitäten zu beobachten, zum anderen sind die zu unterstützenden Geschäftsprozessmodelle oft durch übermäßige Component Interference „entstellt“ worden. In diesem Zusammenhang sind insbesondere die verbreitet festgestellten langen und aus Sicht der untersuchten Geschäftsprozessmodelle unsinnigen Verweisketten als Schwachpunkt zu nennen. Da außerdem die Auswirkungen der Berücksichtigung des globalen Alphabets auf die Suchergebnisse in ihrer Tendenz nicht vorhersehbar sind, legen die in dieser Evaluation erzielten Resultate insgesamt nahe, dass die geschäftsprozessorientierte Komponentensuche auf Basis des globalen Alphabets keine sinnvolle Ergänzung zur Suche auf der Grundlage des lokalen Alphabets darstellt.

Eine Erhöhung der Präzision auf Kosten der Trefferquote lässt sich erreichen, wenn man die geschäftsprozessorientierte Komponentensuche als topologisches Verfahren einsetzt. Berücksichtigt man z. B. nur die Komponenten, die bzgl. der semantischen Abdeckung des betrachteten Geschäftsprozessmodells einen der höchsten drei Werte erreichen, so erhalten wir hinsichtlich des lokalen Alphabets die folgenden Werte für Präzision und Trefferquote:

$$\begin{aligned} \mathit{Präzision}_{top}\mathit{(lokal)} &= \frac{1}{5} \cdot \left(\frac{4}{5} + \frac{3}{3} + \frac{3}{3} + \frac{3}{5} + \frac{1}{1} \right) = 0,88 \\ \mathit{Trefferquote}_{top}\mathit{(lokal)} &= \frac{1}{5} \cdot \left(\frac{4}{4} + \frac{3}{4} + \frac{3}{5} + \frac{3}{4} + \frac{1}{1} \right) = 0,82 \end{aligned}$$

Da nur eine geringe Anzahl von (Prozess-)Komponenten mit einem „interessanten“ Lebenszyklus zur Verfügung gestanden haben, hat der Schwerpunkt dieser Fallstudie auf dem semantischen Vergleich gelegen. Die Suchergebnisse sowie zuvor durchgeführte Experimente haben in diesem Zusammenhang deutlich gezeigt, dass eine „gute“ Terminologie mit einem umfassenden Lexikon, dessen Begriffe konsistent, eindeutig und damit trennscharf definiert sind, eine zentrale Voraussetzung für die geschäftsprozessorientierte Komponentensuche ist. Das terminologische Modell hat sich im Rahmen dieser Evaluation als grundsätzlich ausreichend für die Spezifikation von Terminologien erwiesen, allerdings repräsentiert der Aufbau einer „guten“ Terminologie zweifelsohne eine komplexe Aufgabe. Ein ähnlich komplexes Problem stellt die korrekte Verwendung einer Terminologie zur Beschreibung der fachlichen Semantik von betrieblichen Aktivitäten und Operationen dar.

Ein Ansatz zur Vereinfachung der Nutzung einer Terminologie könnte darin bestehen, nur eine möglichst kompakte Menge von Prädikaten zur Verfügung zu stellen, um dadurch zu klareren Formulierungen der fachlichen Semantik zu gelangen. Ein derart kompaktes Lexikon könnte allgemeine, anwendungsbereichsunabhängige Prädikate wie „erzeugen“, „festlegen“, „ermitteln“, „löschen“, „suchen“, „auswählen“ etc. enthalten, die durch Prädikate zur Beschreibung datenverarbeitender Vorgänge wie z. B. das Herstellen und Lösen von Beziehungen zwischen Informationsobjekten ergänzt werden. Anwendungsbereichsspezifische Aussagen ließen sich dann nur noch durch Kombination dieser Prädikate mit anwendungsbereichsspezifischen Konzepten treffen. Obgleich wir annehmen, dass ein solch kompaktes Lexikon die Spezifikation der fachlichen Semantik vereinfachen und die Effizienz der geschäftsprozessorientierten Komponentensuche steigern kann, ist fraglich, ob

diese Reduzierung des Wortschatzes einen aus Sicht der Geschäftsprozessmodellierung gangbaren Weg darstellt.

Neben der Erstellung einer „guten“ Terminologie besteht für die geschäftsprozessorientierte Komponentensuche allerdings noch weiteres Verbesserungspotential. So hat sich u. a. bei der Analyse der Suchergebnisse zum Geschäftsprozessmodell „Kennzeichen auswählen“ gezeigt, dass die Betrachtung des Datenflusses ein präziseres Bild von der durch eine Komponente angebotenen Unterstützung liefert und damit „korrektere“ Suchergebnisse ermöglicht. Eine Voraussetzung der Berücksichtigung des Datenflusses bei der geschäftsprozessorientierten Komponentensuche ist selbstverständlich dessen Spezifikation im Rahmen der Geschäftsprozessmodellierung. Als Nebeneffekt trägt die explizite Angabe von Informationsobjekten, die im Laufe eines Geschäftsprozesses erzeugt, referenziert, manipuliert oder gelöscht werden, auch zur Reduzierung der Interpretierbarkeit von Geschäftsprozessmodellen bei.

Eine höhere Qualität bei der Zuordnung von Operationen zu betrieblichen Aktivitäten, die durch sie unterstützt werden sollen, lässt sich durch die Bewertung und Minimierung des Abstands zwischen zwei normsprachlichen Sätzen in einer Generalisierungs-/Spezialisierungshierarchie erzielen. Die aktuell verwendete Relation *Impl* (vgl. Definition 8.5) ordnet einer Aktivität eine Menge gleichrangiger Operationen zu. Über den Abstand zwischen normsprachlichen Sätzen könnte eine Rangfolge zwischen den Operationen ermittelt werden, auf deren Grundlage dann ihre Zuordnung zu Aktivitäten beim Semantikvergleich erfolgt.

Weiteres Potential zur Erhöhung der beim Semantikvergleich erzielten Qualität besteht in einer der geschäftsprozessorientierten Komponentensuche vorgelagerten Kategorisierung von Komponenten nach Anwendungsbereichen. Neben der Reduzierung ungeeigneter semantischer Zuordnungen von Operationen zu Aktivitäten trägt diese Kategorisierung auch zur Eingrenzung des Lösungsraums und damit zur Beschleunigung des gesamten Suchverfahrens bei.

Wie wir am Beispiel der Komponente `AdresseBean` und der Aktivität „Datenerfasser ! erfassen Adresse“ gesehen haben, kann die Berücksichtigung von Ganzes/Teile-Beziehungen zwischen Begriffen bei der geschäftsprozessorientierten Komponentensuche in nicht zu vernachlässigendem Maße zur Verbesserung der Suchresultate beitragen. Hier tritt allerdings das Problem auf, dass es ist im Allgemeinen nicht möglich ist, von Aussagen über ein Teil auf Aussagen über das Ganze (oder umgekehrt) zu schließen (vgl. auch [BGMV97]).

10.3 Bewertung von Entwurf und Implementierung

In diesem Abschnitt bewerten wir zunächst den Entwurf der KOSOBAR-Werkzeuge im Hinblick auf ihre Flexibilität, bevor wir auf das Laufzeitverhalten und die Funktionalität ihrer Implementierung eingehen und Verbesserungspotentiale nennen.

Der Entwurf der KOSOBAR-Werkzeuge zeichnet sich durch ein hohes Maß an Anpassbarkeit und Erweiterbarkeit aus, das zum einen auf dem konsequenten Einsatz von Entwurfsmustern und zum anderen auf der durchgängigen Verwendung von Schnittstellen zur Entkopplung von Anbietern und Nutzern von Diensten beruht. Auf übergeordneter Ebene gestattet dabei das Service-To-Worker-Muster [Bie02] die flexible Erweiterung

der Component Matching Engine um neue Kommandos (z. B. zum Veröffentlichen von Komponentenbeschreibungen) sowie die ergebnisabhängige Konfiguration der nach Ausführung dieser Kommandos anzuzeigenden JSPs.

Der in der aktuellen Realisierung der Component Matching Engine für die Erzeugung von Transitionssystemen zuständige `CDLServer` ist aufgrund der durch die Schnittstelle `ComponentServer` erreichten Entkopplung austauschbar, ohne die Anpassung seiner Clients zu erfordern. Mit Hilfe einer anderen Implementierung dieser Schnittstelle könnte die Component Matching Engine z. B. in die Lage versetzt werden, auch nach Komponenten zu suchen, die mit einer anderen Komponentenbeschreibungssprache als CDL spezifiziert sind. Auf analoge Weise können auch die Implementierungen des CDL Component Servers sowie des Terminology Servers ausgetauscht werden. Die konsequente Verwendung des Entwurfsmusters „Fabrikmethode“ [GHJV01] in den jeweiligen APIs trägt hier zudem dazu bei, dass nicht nur die Implementierung der zentralen abstrakten Klassen `Repository` und `Terminology`, sondern auch die der von ihnen erzeugten Objekte für einen Client transparent, d. h. ohne Anpassungen an der Client-Anwendung ausgetauscht werden können. Des Weiteren lässt sich die in der vorliegenden Realisierung eingesetzte Suchstrategie `Simulation` leicht durch eine andere Strategie ersetzen, da die Klasse `ComponentFinder` dem Strategie-Muster [GHJV01] folgend lediglich über eine abstrakte Schnittstelle auf die konkrete Suchstrategie zugreift.

Hinsichtlich der aktuellen Implementierung der KOSOBAR-Werkzeuge ist zunächst einmal festzuhalten, dass diese für den Einsatz im Rahmen der Evaluation unserer Konzepte gute und zuverlässige Unterstützung geboten haben. Ein wesentliches Defizit besteht allerdings im Laufzeitverhalten der Component Matching Engine, das deren praktischem Einsatz entgegensteht. So sind die Antwortzeiten, die beim semantischen Vergleich erreicht werden, insbesondere bei Betrachtung des globalen Alphabets inakzeptabel. Erforderte der Semantikvergleich bzgl. des lokalen Alphabets in der Evaluation im Durchschnitt etwa 4 Minuten, so stieg dieser Wert bei Berücksichtigung des globalen Alphabets auf ca. 32 Minuten an.[1] Hier sind Performancesteigerungen bei der Berechnung des semantischen Netzes, das die Generalisierungs-/Spezialisierungsbeziehungen zwischen Aktivitäten eines Geschäftsprozessmodells und Operationen einer Komponente repräsentiert, dringend vonnöten. Diese könnten z. B. erreicht werden, indem die Berechnung des semantischen Netzes zeitlich vorverlagert wird und normsprachliche Sätze nicht erst bei der Komponentensuche, sondern bereits bei deren Erzeugung durch den Terminology Server in eine entsprechende Generalisierungs-/Spezialisierungshierarchie eingeordnet werden. Dazu ist die Integration eines Klassifizierungswerkzeugs wie z. B. das in dem Softwareinformationssystem LaSSIE (vgl. Abschnitt 4.3.2) eingesetzte KANDOR in den Terminology Server erforderlich.

Während das Laufzeitverhalten des Protokollvergleichs bei Berücksichtigung des lokalen Alphabets noch akzeptabel ist, muss es bei Betrachtung des globalen Alphabets ebenfalls als kritisch bewertet werden. Da in diesem Fall weniger Aktionen in den untersuchten Geschäftsprozessmodellen und Komponentenbeschreibungen bzw. den ihnen zugrunde liegenden Transitionssystemen versteckt werden, ist im Zuge des Protokoll-

[1] Die Messwerte wurden auf einem PC mit Pentium-III-Prozessor (850 MHz) und 576 MB Hauptspeicher unter Windows 2000 ermittelt, der zwecks Terminologieverwaltung einen Datenbank-Server mit zwei Pentium-III-Prozessoren (850 MHz) und 1024 MB Hauptspeicher genutzt hat.

vergleichs eine höhere Anzahl (stabiler) Zustände erreichbar, die das Laufzeitverhalten polynomiell beeinflusst (vgl. hierzu die Ausführungen zur Komplexität des Protokollvergleichs in Abschnitt 7.5.2). So wurde die Mehrheit der Anfragen bei Betrachtung des lokalen Alphabets im einstelligen Sekundenbereich beantwortet und ein Durchschnittswert von ca. 30 Sekunden erzielt. Bei Berücksichtigung des globalen Alphabets erhöhte sich die durchschnittliche Dauer des Protokollvergleichs auf ungefähr $1,5$ Minuten, wobei Maximalwerte von bis zu 5 Minuten erreicht wurden. In diesem Zusammenhang ist der Einsatz von Metriken zur Bewertung von Suchpfaden sowie darauf aufbauend die Verwendung von Heuristiken, mit denen „schlechte“ Suchpfade ausgeschlossen werden können, zu erwägen. Da die in der Component Matching Engine aktuell eingesetzte Suchstrategie zum einen auf einem einfachen Backtracking-Algorithmus basiert und zum anderen wie oben beschrieben austauschbar ist, lassen sich derartige Maßnahmen auf einfache Weise integrieren.

Die derzeitige Implementierung des Terminology Managers nutzt eine Baumstruktur zur Repräsentation von Generalisierungs-/Spezialisierungsbeziehungen zwischen den Begriffen einer Terminologie (vgl. Abbildung 9.11). Obgleich unser in Abschnitt 8.3.2 vorgestelltes terminologisches Modell die Modellierung von Begriffen gestattet, die mehr als eine Generalisierung aufweisen, wird diese Möglichkeit aufgrund dieser Realisierungsentscheidung praktisch nicht unterstützt. Eine entsprechende Anpassung der Implementierung ist als sinnvoll einzustufen, um z. B. den Begriff „Halter“ sowohl als natürliche als auch als juristische Person modellieren zu können. Da die beschriebenen Änderungen nur die Benutzungsoberfläche betreffen, schätzen wir den erforderlichen Anpassungsaufwand als gering ein.

Mit Blick auf die Implementierung des Component Retrieval Interfaces ist festzuhalten, dass die graphische Präsentation der durch eine Komponente angebotenen Unterstützung eines Geschäftsprozessmodells sinnvoll zur Bewertung der Komponente durch einen Fachexperten und damit zur Entscheidungsfindung bei der Make-or-Buy-Frage beitragen kann. Diese Präsentation der Suchergebnisse kann durch die Darstellung der von einer Komponente tatsächlich „ausgeführten“ Prozesse sinnvoll ergänzt werden. Der Umfang der von der Component Matching Engine akzeptierten Component Interference könnte auf diese Weise explizit gemacht werden und in die Bewertung der Komponente einfließen. Da die für diese Ergänzung erforderlichen Informationen der Component Matching Engine vorliegen, lässt sich eine entsprechende Anpassung auf der Grundlage der Graphenbibliothek *Grappa* mit vergleichsweise niedrigem Aufwand realisieren.

10.4 Zusammenfassung

In diesem Kapitel haben wir die Evaluation der Konzepte zur geschäftsprozessorientierten Komponentensuche und ihrer prototypischen Umsetzung dargestellt. Ausgehend von der Einführung einer Fallstudie aus dem Anwendungsbereich „Kfz-Zulassungswesen“, anhand der wir die Evaluation durchgeführt haben, haben wir zunächst die bei der Komponentensuche beobachteten Ergebnisse vorgestellt und im Hinblick auf Stärken und Schwächen des Ansatzes bewertet. Da uns zu wenige Komponenten mit einem „interessanten“ Lebenszyklus zur Verfügung standen, lag der Schwerpunkt der betrachteten Fallstudie und damit auch der Evaluation auf dem semantischen Vergleich. Im Anschluss haben wir den

Entwurf und die Implementierung der KOSOBAR-Werkzeuge hinsichtlich Flexibilität, Laufzeitverhalten und Funktionalität bewertet.

Als Fazit der Evaluation lässt sich zusammenfassend festhalten, dass die geschäftsprozessorientierte Komponentensuche bzgl. der im Rahmen der Fallstudie betrachteten Daten vielversprechende Werte für Präzision und Trefferquote gezeigt hat, wenn lediglich das lokale Alphabet betrachtet wurde. Durch Berücksichtigung des globalen Alphabets konnten die Resultate weder in qualitativer noch in quantitativer Hinsicht verbessert werden. Darüber hinaus ist deutlich geworden, dass der semantische Vergleich einerseits großen Einfluss auf die Qualität der Suchergebnisse hat, andererseits allerdings auch große Probleme bereitet. Sowohl die Erstellung einer „guten“ Terminologie als auch deren korrekte Verwendung stellen in diesem Zusammenhang komplexe Aufgaben dar. Der Protokollvergleich, über dessen Auswirkungen auf die Qualität der Suchergebnisse wir aufgrund der eingangs genannten Einschränkungen der Fallstudie keine eindeutigen Aussagen ermitteln konnten, ist dagegen mit den verfügbaren formalen Mitteln (Repräsentationsformalismen, Algorithmen) auf einfachere Weise beherrschbar.

Eine Erhöhung von Präzision und Trefferquote der geschäftsprozessorientierten Komponentensuche lässt sich durch die Betrachtung des Datenflusses beim Protokollvergleich, die Bewertung und Minimierung des semantischen Abstands zwischen Aktivitäten und Operationen sowie die Berücksichtigung von Ganzes/Teile-Beziehungen beim Semantikvergleich erreichen. Dringend notwendige Verbesserungen des Laufzeitverhaltens der KOSOBAR-Werkzeuge zur geschäftsprozessorientierten Komponentensuche lassen sich insbesondere durch eine zeitlich vorverlagerte Berechnung des semantischen Netzes zwischen den Aktivitäten eines Geschäftsprozessmodells und den Operationen einer Komponente sowie den Einsatz von Metriken und Heuristiken zur Bewertung bzw. Auswahl von Suchpfaden beim Protokollvergleich erzielen.

Die in diesem Kapitel vorgestellte Evaluation kann nur einen ersten Anhaltspunkt für die Stärken und Schwächen des Ansatzes darstellen. Um verlässlichere Aussagen über Präzision und Trefferquote der geschäftsprozessorientierten Komponentensuche zu erhalten, sind weitere Evaluationen unabdingbar. Dabei sollte im Anschluss an die Durchführung von Maßnahmen zur Verbesserung des Laufzeitverhaltens eine möglichst umfangreiche Menge an Geschäftsprozessmodellen und (domänenspezifischen und -unabhängigen) Komponenten betrachtet werden. Interessant ist darüber hinaus eine Analyse der qualitativen Veränderung der Suchergebnisse, wenn – anders als in der hier vorgestellten Evaluation – Geschäftsprozessmodelle und Komponenten durch verschiedene Personen beschrieben werden.

Teil IV

Zusammenfassung und Ausblick

Kapitel 11

Zusammenfassung und Ausblick

In diesem Kapitel schließen wir die vorliegende Arbeit mit einer Zusammenfassung und einem Ausblick ab. Abschnitt 11.1 stellt zunächst die Ziele dieser Arbeit, den verfolgten Ansatz sowie die erreichten Ergebnisse rückblickend dar. Künftige Herausforderungen und weiterer Forschungsbedarf im Kontext der komponentenbasierten Softwareentwicklung werden in Abschnitt 11.2 diskutiert.

11.1 Zusammenfassung

Seit dem Aufkommen von Komponententechnologien wie COM+ und Enterprise JavaBeans (EJB) gegen Ende der 90er Jahre setzt sich das Paradigma der komponentenbasierten Softwareentwicklung in zunehmendem Maße durch. Der Entwicklung von Komponentensoftware liegt dabei die Idealvorstellung zugrunde, die Vorteile von Individual- und Standardsoftware miteinander zu kombinieren, indem Software durch die kundenindividuelle Komposition und Anpassung standardisierter Komponenten „konstruiert“ wird.

Zu den zentralen Merkmalen komponentenbasierter Softwareentwicklungsprozesse gehören die Suche nach wiederverwendbaren Komponenten, die auf Komponentenmärkten angeboten werden, sowie die sich daran anschließende Make-or-Buy-Entscheidung. Aktuelle Komponentenmärkte unterstützen die Komponentensuche nur in unzureichendem Maße, da verfügbare Komponenten zumeist uneinheitlich, unvollständig und missverständlich beschrieben werden, nur eingeschränkte Möglichkeiten zur Spezifikation fachlicher Anforderungen an gesuchte Komponenten angeboten werden und lediglich einfache Suchverfahren wie z. B. Stichwortsuche in Freitextbeschreibungen Verwendung finden.

Aufgrund dieser Defizite erzielen aktuelle Komponentenmärkte gemessen an den tatsächlichen fachlichen Anforderungen eines Verwenders nur eine niedrige Präzision. Da insbesondere Anforderungen, die in Form von Geschäftsprozessmodellen festgehalten werden, keine Berücksichtigung bei der Komponentensuche finden, resultiert vielfach die Notwendigkeit der Anpassung von Geschäftsprozessen an die gefundenen (und eingesetzten) Komponenten. Das übergeordnete Ziel dieser Arbeit bestand daher darin, die Präzision der Komponentensuche zu erhöhen und die Bewertung gefundener Komponenten zu unterstützen, indem die in Geschäftsprozessmodellen festgehaltenen fachlichen Anforderungen bei der Komponentensuche explizit berücksichtigt werden.

Der Kern des in dieser Arbeit vorgestellten Ansatzes besteht in der Übertragung des Prinzips des Behavioural Subtyping auf die Komponentensuche. Über die Bestimmung von Subtyp-Beziehungen zwischen Geschäftsprozessmodellen und Komponentenbeschreibungen werden dabei Komponenten identifiziert, die die Ausführung (von Teilen) der spezifizierten Geschäftsprozesse durch geeignete Funktionalität unterstützen. Dazu werden auf der Protokollebene die durch ein Geschäftsprozessmodell festgelegte Ablaufstruktur mit dem Interaktionsprotokoll einer Komponente verglichen und auf der Semantikebene die fachliche Verträglichkeit einzelner Aktivitäten eines Geschäftsprozessmodells mit Operationen einer Komponente überprüft. Damit haben sich die folgenden Einzelziele dieser Arbeit ergeben:

1. Konzipierung einer Komponentenbeschreibungssprache, die neben der Spezifikation struktureller Aspekte auch die Beschreibung des beobachtbaren Verhaltens einer Komponente gestattet,
2. Entwicklung einer Behavioural-Subtyping-Relation für den formalen Vergleich der durch Geschäftsprozessmodelle und Komponentenbeschreibungen spezifizierten Protokolle sowie
3. Angabe einer Repräsentationsform für die fachliche Semantik von Aktivitäten und Operationen, die auch für Fachexperten verständlich ist und semantische Vergleiche im Sinne des Subtyping-Prinzips zulässt.

Zur Erläuterung der Beiträge, die die vorliegende Arbeit im Hinblick auf diese Zielsetzungen geliefert hat, fassen wir im Folgenden Aufbau und Inhalt der Arbeit noch einmal kurz zusammen.

Ausgehend von einer Einführung in das Paradigma der komponentenbasierten Softwareentwicklung haben wir in Kapitel 1 die Motivation und Zielsetzung unserer Arbeit herausgearbeitet. In den Kapiteln 2 bis 4 haben wir anschließend die zentralen Begriffe der Geschäftsprozessmodellierung und der Komponentensoftware eingeführt sowie einen Überblick über den State of the Art auf dem Gebiet der Komponentensuche gegeben.

Nach einer Eingrenzung von Ziel und Gegenstand der geschäftsprozessorientierten Komponentensuche haben wir in Kapitel 5 eine Interpretation der geschäftsprozessorientierten Komponentensuche als Subtyping-Problem vorgestellt und die Übertragung wesentlicher Gedanken des Behavioural Subtyping auf die Komponentensuche motiviert. Auf der Grundlage dieser Überlegungen konnten wir schließlich den Makroprozess der geschäftsprozessorientierten Komponentensuche skizzieren und den Aufbau des zweiten Teils der Arbeit umreißen, der die von uns entwickelten Konzepte in den Kapiteln 6 bis 8 präsentiert.

In Kapitel 6 haben wir zunächst die Komponentenbeschreibungssprache *Component Description Language (CDL)* eingeführt und im Anschluss unsere Entscheidung für die Betrachtung linearer Prozessmodelle als Sprache zur Spezifikation von Suchanfragen begründet. Beim Entwurf der CDL standen zum einen die integrierte Beschreibung der Struktur und des Verhaltens von Komponenten im Vordergrund, zum anderen bestand eine zentrale Anforderung darin, fachlich orientierte Komponentenbeschreibungen im Sinne „komponentenorientierter Softwarereferenzmodelle“ zu ermöglichen, die auch für Fachexperten verständlich sind.

In Kapitel 7 haben wir die Subtyping-Relation *Match* für den formalen Vergleich von linearen Prozessmodellen und Komponentenbeschreibungen entwickelt und einen einfachen Algorithmus zur Berechnung dieser Relation angegeben. Dieser Algorithmus erzeugt Berechnungsaufwand in der Größenordnung $\mathcal{O}(n^4)$, wobei n die maximale Anzahl der Zustände des Geschäftsprozessmodells und der Komponentenbeschreibung angibt. Bei der Entwicklung von Match haben wir auf die Betrachtung des Datenflusses in Geschäftsprozessen und Komponenten sowie die Berücksichtigung der Parametrisierung von Zuständen und die Auswertung von Bedingungen verzichtet.

Konzepte zur Repräsentation der fachlichen Semantik von Aktivitäten und Operationen sowie deren Vergleich haben wir in Kapitel 8 eingeführt. Wir haben dazu zunächst die aus Sicht der geschäftsprozessorientierten Komponentensuche relevanten Eigenschaften von Geschäftsprozessmodellen und Komponentenbeschreibungen normsprachlich rekonstruiert. Auf der Grundlage der resultierenden Normsprache haben wir anschließend die fachliche Semantik von Aktivitäten und Operationen formal definiert, die wir für den semantischen Vergleich von Geschäftsprozessmodellen und Komponentenbeschreibungen durch die Implementierungsrelation *Impl* heranziehen. Dem gewählten Subtyping-Ansatz folgend beschränkt sich der Semantikvergleich auf die Betrachtung von Generalisierungs-/Spezialisierungsbeziehungen.

Gegenstand des dritten Teils unserer Arbeit war die Beschreibung der prototypischen Umsetzung der Konzepte zur geschäftsprozessorientierten Komponentensuche und deren Evaluation. In Kapitel 9 haben wir dazu ausgewählte Aspekte des Entwurfs und der Implementierung von Werkzeugen zur Verwaltung von Komponenten und Terminologien sowie zur Komponentensuche vorgestellt. Ausgehend von einer abstrakten Systemarchitektur haben wir zunächst den Entwurf der wichtigsten Werkzeuge dieser Architektur dokumentiert. Nach der Konkretisierung der Systemarchitektur haben wir schließlich die realisierten Werkzeuge übersichtsartig vorgestellt.

In Kapitel 10 haben wir die Evaluation der geschäftsprozessorientierten Komponentensuche anhand einer Fallstudie aus dem Anwendungsbereich „Kfz-Zulassungswesen“ vorgestellt. Obgleich Art und Umfang der Fallstudie noch keine belastbaren Schlüsse über die Effektivität der geschäftsprozessorientierten Komponentensuche zuließen, konnten wir vielversprechende Resultate ermitteln. Als Maßnahmen zur Steigerung der Präzision haben wir insbesondere die Betrachtung des Datenflusses beim Protokollvergleich sowie die Bewertung und Minimierung des semantischen Abstands zwischen Aktivitäten und Operationen beim Semantikvergleich identifiziert. Die Trefferquote lässt sich durch die zusätzliche Berücksichtigung von Ganzes/Teile-Beziehungen zwischen Begriffen erhöhen. Während sich der Entwurf der realisierten Werkzeuge insbesondere durch seine Flexibilität auszeichnet, genügt das Laufzeitverhalten der prototypischen Implementierung noch nicht den Anforderungen eines praktischen Einsatzes.

Wir haben die gesteckten Einzelziele dieser Arbeit durch die Entwicklung der *CDL* (Kapitel 6), die Konzipierung der Behavioural-Subtyping-Relation *Match* (Kapitel 7) und die Definition einer Spezialisierungsrelation *Impl* zum Vergleich der fachlichen Semantik von Aktivitäten und Operationen (Kapitel 8) erreicht. Das übergeordnete Ziel der Arbeit, die Präzision der Komponentensuche durch explizite Berücksichtigung der in Form von Geschäftsprozessmodellen festgehaltenen fachlichen Anforderungen zu erhöhen, konnte dagegen noch nicht vollständig erreicht werden. Zwar ist ein quantitativer Vergleich der

erzielten Präzisionswerte mit entsprechenden Kennzahlen der in Kapitel 4 vorgestellten Suchverfahren nicht sinnvoll, da zum einen eine ernsthafte empirische Absicherung der von uns ermittelten Werte mit einem größeren, praxisrelevanten Beispiel noch aussteht und zum anderen die Kennzahlen der alternativen Suchverfahren – sofern solche Angaben verfügbar sind – nicht mit vergleichbaren Anforderungen und Komponenten ermittelt wurden. Eine qualitative Bewertung der in der Evaluation erzielten Resultate hat allerdings Verbesserungspotentiale des Ansatzes hinsichtlich Präzision und Trefferquote aufgezeigt.

11.2 Ausblick

Das Ziel, komplexe Softwaresysteme auf der Grundlage vorgefertigter Komponenten „konstruieren“ zu können, wird in der Informatik inzwischen seit mehr als drei Jahrzehnten intensiv verfolgt. Trotz deutlich erkennbarer Fortschritte, die in den letzten Jahre insbesondere durch Komponententechnologien wie COM+, EJB und zuletzt .NET und deren Akzeptanz in der Softwareentwicklung erreicht wurden, muss konstatiert werden, dass die komponentenbasierte Softwareentwicklung bei weitem noch nicht die Reife von Entwicklungs- und Herstellungsprozessen anderer Ingenieurdisziplinen erreicht hat.

Als eine der zentralen Herausforderungen, der sich die komponentenbasierte Softwareentwicklung in den nächsten Jahren stellen muss, sehen wir auf Seiten der Komponentenverwender die organisatorische „Installation“ inter- und intraorganisationaler Wiederverwendung an. Dies kann durch verschiedenartige organisatorische Maßnahmen wie z. B. die Einführung von Entwicklungsprozessen mit speziellen Wiederverwendungsaktivitäten („develop with reuse“, „develop for reuse“), die Definition spezifischer Rollen, die mit der Entwicklung, Qualitätssicherung und Verwaltung wiederverwendbarer Komponenten betraut werden, oder die Unterstützung der Softwareentwicklung durch geeignete Werkzeuge wie z. B. Komponenten-Repositories erreicht werden.

Auf Seiten der Komponentenhersteller besteht eine zweite zentrale Herausforderung der komponentenbasierten Softwareentwicklung im Wandel des Verhältnisses zu ihren potentiellen Kunden. Anders als in „reifen“ Ingenieurdisziplinen wie z. B. der Elektrotechnik oder dem Bauwesen, wo Komponenten mitsamt detaillierter, standardisierter Beschreibungen ihrer relevanten Eigenschaften in Produktkatalogen angeboten werden, werden die für die Bewertung einer Komponente relevanten Informationen in der Softwaretechnik vielfach gar nicht, nur sehr unvollständig oder lediglich in „verschleierter“ und damit nicht vergleichbarer Form veröffentlicht. Der weitere Reifeprozess der komponentenbasierten Softwareentwicklung hängt in nicht unerheblichem Maße davon ab, dass Komponentenhersteller Komponentenbeschreibungen als konstitutives Element der von ihnen angebotenen Komponenten begreifen und einen offeneren, offensiveren Umgang mit diesen Informationen pflegen.

Aufbauend auf dem beschriebenen Wandel stellt die Komponentensuche auf inter- und intraorganisationalen Komponentenmärkten einen Schlüssel für die Akzeptanz der organisatorischen Maßnahmen dar, die wir zur „Installation“ der Wiederverwendung in Organisationen gefordert haben. Aufgabe der Komponentensuche ist das Schließen der konzeptuellen Lücke zwischen Anforderungsdokumenten, die neben Technologieexperten vielfach auch von Fachexperten formuliert werden (Situationsmodell), und den Beschrei-

bungen verfügbarer Softwarelösungen (Systemmodell). Eine Voraussetzung für die Effektivität von Suchverfahren wie der in dieser Arbeit vorgestellten geschäftsprozessorientierten Komponentensuche ist dabei die Existenz geeigneter Notations- und Fachstandards zur Beschreibung von Komponenten. In Forschung und Praxis sind verschiedene Vorschläge zur syntaktischen und semantischen Beschreibung von Komponenten entwickelt worden, von denen sich allerdings bislang keiner durchsetzen konnte.

Notationsstandards müssen die umfassende, eindeutige und nachprüfbare Beschreibung von Komponenten unterstützen, wobei insbesondere die Rolle der Fachexperten in der komponentenbasierten Softwareentwicklung zu berücksichtigen ist. Die von uns vorgeschlagene Komponentenbeschreibungssprache CDL liefert u. a. durch die Spezifikation des von einer Komponente unterstützten Protokolls sowie der fachlichen Semantik ihrer Operationen einen entsprechenden Beitrag. Fachstandards zur Spezifikation inhaltlicher Bezüge ergänzen die auf syntaktische Aspekte ausgerichteten Notationsstandards, um die Verständlichkeit und Vergleichbarkeit von Komponentenbeschreibungen zu fördern. Dabei können Unternehmensontologien ein Mittel zur Strukturierung solcher Fachstandards darstellen, deren Inhalte durch Lexika wie z. B. *WordNet* definiert sind. Das in dieser Arbeit entwickelte terminologische Modell stellt einen Beitrag zur Strukturierung von Fachstandards dar.

Die Entwicklung des Ansatzes der geschäftsprozessorientierten Komponentensuche, wie er in dieser Arbeit präsentiert wurde, kann sicherlich nicht als abgeschlossen angesehen werden. Neben einer vertieften Evaluation der Konzepte anhand einer umfangreicheren Auswahl an Geschäftsprozessmodellen und Komponentenbeschreibungen ergeben sich mit der Weiterentwicklung der CDL sowie des Protokoll- und Semantikvergleichs drei Forschungsbereiche im unmittelbaren Kontext dieser Arbeit:

- *Weiterentwicklung der CDL*: Auf Seiten der CDL ist zunächst einmal eine Erhöhung des Detaillierungsgrades von Verhaltensbeschreibungen durch die Erweiterung des `existing`-Operators um die explizite Spezifikation der möglichen Zustände referenzierter Geschäftsobjekte wünschenswert. Diese Erweiterung kann z. B. dazu beitragen, die Semantik von Finder-Operationen präziser zu formulieren. Des Weiteren stellt die Frage nach der Erfüllung eines Kontrakts, der eine untere Grenze für das aufrufbare Verhalten definiert, durch eine Komponente, deren Beschreibung eine obere Grenze für das beobachtbare Verhalten spezifiziert, ein interessantes Forschungsgebiet dar, das wir im Rahmen dieser Arbeit nicht untersucht haben. Ein Forschungsthema, das wir in dieser Arbeit nur gestreift haben, ist die Beschreibung der Konfigurationsalternativen und Konfigurationen von (komplexen) Komponenten. In diesem Zusammenhang erscheint aus unserer Sicht insbesondere die Repräsentation der zu erwartenden Auswirkungen von Konfigurationsentscheidungen auf das Verhalten einer Komponente von Interesse.

- *Präzisions- und Effizienzsteigerung beim Protokollvergleich*: Die Evaluation unserer Konzepte zur geschäftsprozessorientierten Komponentensuche hat Hinweise dafür geliefert, dass hinsichtlich des Protokollvergleichs Steigerungen der Präzision sowie der Effizienz anzustreben sind. Die Erhöhung der Präzision lässt sich dabei zunächst einmal durch die Betrachtung des Datenflusses, d. h. die Erzeugung und Referenzierung von Geschäftsobjekten in Geschäftsprozessmodellen und Kompo-

nentenbeschreibungen erreichen. Aufbauend auf dieser Ergänzung könnte in einem zweiten Schritt analysiert werden, inwieweit die zusätzliche Berücksichtigung der Parameter von Zuständen in Komponentenbeschreibungen zur weiteren Steigerung der Präzision beiträgt. In diesem Zusammenhang ist ggf. der Einsatz von Techniken der Datenabstraktion zu erwägen, um Verhaltensbeschreibungen mit unendlichen Zustandsräumen effizient miteinander vergleichen zu können. Zur Steigerung der Effizienz des Protokollvergleichs bieten sich mit der Entwicklung eines effizienteren Algorithmus sowie dessen Ergänzung um Metriken zur Bewertung von Suchpfaden und „intelligenten“ Heuristiken zur Auswahl „guter“ Suchpfade zwei komplementäre Maßnahmen an, die Gegenstand künftiger Forschungsarbeiten sein könnten.

- *Weiterentwicklung des Semantikvergleichs*: Im Rahmen der Evaluation des semantischen Vergleichs haben wir bereits auf die Berücksichtigung von Ganzes/Teile-Beziehungen zwischen Begriffen als Ansatz zur Steigerung der Trefferquote der geschäftsprozessorientierten Komponentensuche sowie die damit verbundenen grundsätzlichen Probleme hingewiesen. Eine weitere interessante Fragestellung für künftige Forschungsarbeiten zum Semantikvergleich könnte in der Betrachtung von Verfeinerungsbeziehungen zwischen Aktivitäten und Operationen bestehen. Unter Verwendung von Techniken des Action Refinement könnte in diesem Zusammenhang untersucht werden, ob sich die Ausführung einer betrieblichen Aktivität im Sinne der hierarchischen Prozessmodellierung durch Sequenzen feinerer Operationen unterstützen lässt. Im Hinblick auf das in dieser Arbeit vorgestellte terminologische Modell sind Konzepte zur modularen Strukturierung von Terminologien zu entwickeln. Mit diesen Konzepten soll der inkrementelle Aufbau interoperabler Terminologien unterstützt werden, die durch Ergänzung einer allgemeinen Basisterminologie mit organisations- oder domänenspezifischen Erweiterungsmodulen entstehen.

In dieser Arbeit haben wir uns auf die Betrachtung von Geschäftsprozessen konzentriert, die innerhalb der Grenzen eines Unternehmens ausgeführt werden. Eine nahe liegende Erweiterung durch zukünftige Arbeiten besteht daher darin, die geschäftsprozessorientierte Komponentensuche im Kontext unternehmensübergreifender Geschäftsprozesse zu betrachten. In diesem Zusammenhang stellt insbesondere die Untersuchung der Übertragbarkeit der vorgestellten Konzepte zur geschäftsprozessorientierten Komponentensuche auf die Suche und Komposition von Web Services zur Abwicklung unternehmensübergreifender Geschäftsprozesse ein interessantes Gebiet für künftige Forschungsarbeiten dar.

Anhänge und Verzeichnisse

Anhang A

Grammatik der Component Description Language (CDL)

Nachfolgend ist die Grammatik der Component Description Language (CDL) in EBNF-Notation angegeben, auf deren Grundlage der Parser-Generator JavaCC einen CDL-Parser erzeugt hat. Die Semantik der CDL wird in Abschnitt 6.1 informell beschrieben.

system ::=
(*contract* | *component* | *complex-comp*)+.

contract ::=
`contract` *path* `{`
(*slot*)*
(*objecttype*)*
(*procedure*)*
[*c-behaviour i-behaviour*] `};` .

component ::=
`component` *path* [*supports*] `{`
(*slot*)*
(*class*)*
(*procedure*)*
[*c-behaviour i-behaviour*] `};` .

complex-comp ::=
`complex` *path* `=` *path slotcomponents*`;`.

supports ::=
`supports` *path* (`,` *path*)*.

slot ::=
`slot` *path id*`;` .

objecttype ::=
`objecttype` *id* `{`
(*procedure*)*
[*c-behaviour i-behaviour*] `};` .

class ::=
`class` *id* `{`
(*procedure*)*
[*c-behaviour i-behaviour*] `};` .

slotcomponents ::=
`(` *path* (`,` *path*)* `)` .

procedure ::=
id `(` [*fparas*] `)` [`{` *procbody* `}`] `;` .

procbody ::=
command `;` *procbody*
| *return* `;` .

command ::=
send
| *new*
| *existing*
| *choice*
| *repeatable-block* .

repeatable-block ::=
`{` (*command* `;`)+ (`}+` | `}*`).

return ::=
`return` [`(` [*aparas*] `)`].

new ::=
`new` `(` *path id* `)`.

existing ::=
`existing` `(` *path id* `)`.

choice ::=
`choice {` (`+` [`[` *condition* `]`] *choice-body*)+ `}`.

choice-body ::=
command `;` [*choice-body*]
| *return* `;`.

c-behaviour ::=
`created() =` *send-transitions* `;`.

i-behaviour ::=
`initialized (` [*sparas*] `) =` *statedescr* `;`
[*stdescriptions*].

stdescriptions ::=
state `=` *statedescr* `;` [*stdescriptions*] .

state ::=
id `(` [*sparas*] `)`.

statedescr ::=
(*callevent* | *recv-transitions*)
(`+` (*callevent* | *recv-transitions*))*.

send-transitions ::=
`initialized(` [*sparas*] `)`
| *send* `.` *send-transitions*.

recv-transitions ::=
recv-sequence `.` *statechange*
| `( )`.

recv-sequence ::=
recv [`.` *recv-sequence*]
| *split-join-block* [`.` *recv-sequence*].

split-join-block ::=
`(` *recv-sequence* (`+` *recv-sequence*)+ `)`.

callevent ::=
`on` *event* `if` *simple-condition* `do` [*recv-sequence* `.`] *statechange*

statechange ::=
`initialized()`
| `( )`
| *id* `(` [*aparas*] `)`.

recv ::=
id `?` `(` [*aparas*] `)`.

send ::=
(`parent` | `self` | *id*) `->` *id* `!` `(` [*aparas*] `)`.

event ::=
recv.

condition ::=
simple-condition | *send*.

simple-condition ::=
expression operator expression.

operator ::=
`==`
| `<`
| `>`
| `<=`
| `>=`
| `not` .

expression ::=
id
| *num*.

fparas ::=
fpara (`,` *fpara*)*.

fpara ::=
(`in` | `out` | `inout`) *id* `:` *type*.

aparas ::=
apara (`,` *apara*)*.

apara ::=
`*`
| (*id* | *num*) ((`+` | `-` | `*`) (*id* | *num*))*.

sparas ::=
spara (`,` *spara*)*.

spara ::=
id `:` *type*.

type ::=
`boolean`
| `char`
| `byte`
| `short`
| `int`
| `long`
| `float`
| `double`
| `String`
| `Object`
| *set*
| *path*.

set ::=
`Set<`*type*`>`.

path ::=
id (`.` *path*)*.

id ::=
[`a-z`, `A-Z`] ([`a-z`, `A-Z`, `0-9`, `_`])*.

num ::=
([`0-9`])+.

Anhang B

Document Type Definition linearer Prozessmodelle

Die nachfolgend angegebene Document Type Definition (DTD) linearer Prozessmodelle, deren syntaktische Korrektheit mit dem Werkzeug *xmlspy* geprüft wurde, definiert strukturelle Anforderungen an die XML-Repräsentation linearer Prozessmodelle. Diese ist Grundlage der Formulierung von Suchanfragen, die von der Component Matching Engine (vgl. Abschnitt 9.2.3) verarbeitet werden.

```
<!ENTITY % block "(activityInstanceReference | generic | epsilon
                  | sequence | parallel | alternative | loop)">
<!ENTITY % expression "(baseExpression | assembledExpression)">
<!ELEMENT processSchema (activitySet, subSchema+)>
<!ELEMENT activitySet (activity*)>
<!ELEMENT activity (key, name, description, class, simulation,
                    inParameterSet, outParameterSet, roles)>
<!ELEMENT key (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT simulation ((mean, deviation) | (minTime, maxTime)
                      | (duration))>
<!ELEMENT mean (#PCDATA)>
<!ELEMENT deviation (#PCDATA)>
<!ELEMENT minTime (#PCDATA)>
<!ELEMENT maxTime (#PCDATA)>
<!ELEMENT duration (#PCDATA)>
<!ELEMENT inParameterSet (inParameter*)>
<!ELEMENT inParameter (#PCDATA)>
<!ELEMENT outParameterSet (outParameter*)>
<!ELEMENT outParameter (#PCDATA)>
<!ELEMENT roles (role*)>
<!ELEMENT role (#PCDATA)>
<!ELEMENT subSchema (name, description, activityInstanceSet, structure)>
<!ELEMENT activityInstanceSet (activityInstance*)>
<!ELEMENT activityInstance EMPTY>
<!ELEMENT structure (%block;)>
<!ELEMENT activityInstanceReference EMPTY>
<!ELEMENT generic EMPTY>
```

```
<!ELEMENT epsilon EMPTY>
<!ELEMENT sequence ((%block;),(%block;)+)>
<!ELEMENT parallel ((%block;),(%block;)+)>
<!ELEMENT alternative (rules, (%block;),(%block;)+)>
<!ELEMENT loop (rule, (%block;))>
<!ELEMENT rule (decision)?>
<!ELEMENT rules (decision)*>
<!ELEMENT decision (%expression;)>
<!ELEMENT baseExpression (feature, value)>
<!ELEMENT assembledExpression ((%expression;), (%expression;)+)>
<!ELEMENT feature (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ATTLIST processSchema rootSchema IDREF #REQUIRED>
<!ATTLIST activity id ID #REQUIRED>
<!ATTLIST simulation type (fix | uniform | normal) #REQUIRED>
<!ATTLIST subSchema id ID #REQUIRED>
<!ATTLIST activityInstance activity IDREF #REQUIRED>
<!ATTLIST activityInstance id ID #REQUIRED>
<!ATTLIST activityInstanceReference activityInstance IDREF #REQUIRED>
<!ATTLIST assembledExpression type (conjunctive | disjunctive) #REQUIRED>
```

Anhang C

Evaluation

Dieser Anhang fasst die im Rahmen der Evaluation der Konzepte zur geschäftsprozessorientierten Komponentensuche verwendeten Daten zusammen. Grundlage der Beschreibung der untersuchten Komponenten und Geschäftsprozesse ist die in Abschnitt C.1 dokumentierte Terminologie. Die als Suchanfragen eingesetzten Geschäftsprozessmodelle werden in Abschnitt C.2 dargestellt, während die betrachteten Komponenten durch die CDL-Beschreibungen aus Abschnitt C.3 spezifiziert wurden.

C.1 Terminologie

In diesem Abschnitt skizzieren wir die bei der Evaluation eingesetzte Terminologie. Abbildung C.1 illustriert zunächst die verwendeten Aktivitäten in Form eines Klassendiagramms. Dabei kennzeichnen wir abstrakte Begriffe, die nicht für die Bildung von normsprachlichen Sätzen zur Verfügung stehen, durch kursive Schrift.

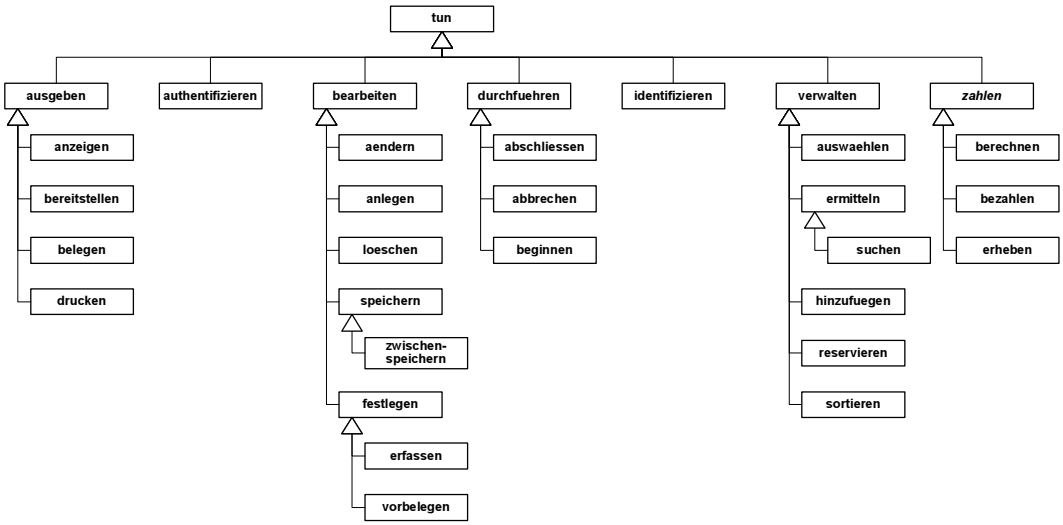


Abbildung C.1: Terminologie (Aktivitäten)

Die Abbildungen C.2 und C.3 stellen die Generalisierungsbeziehungen zwischen den genutzten Konzepten wiederum in Gestalt von Klassendiagrammen graphisch dar.

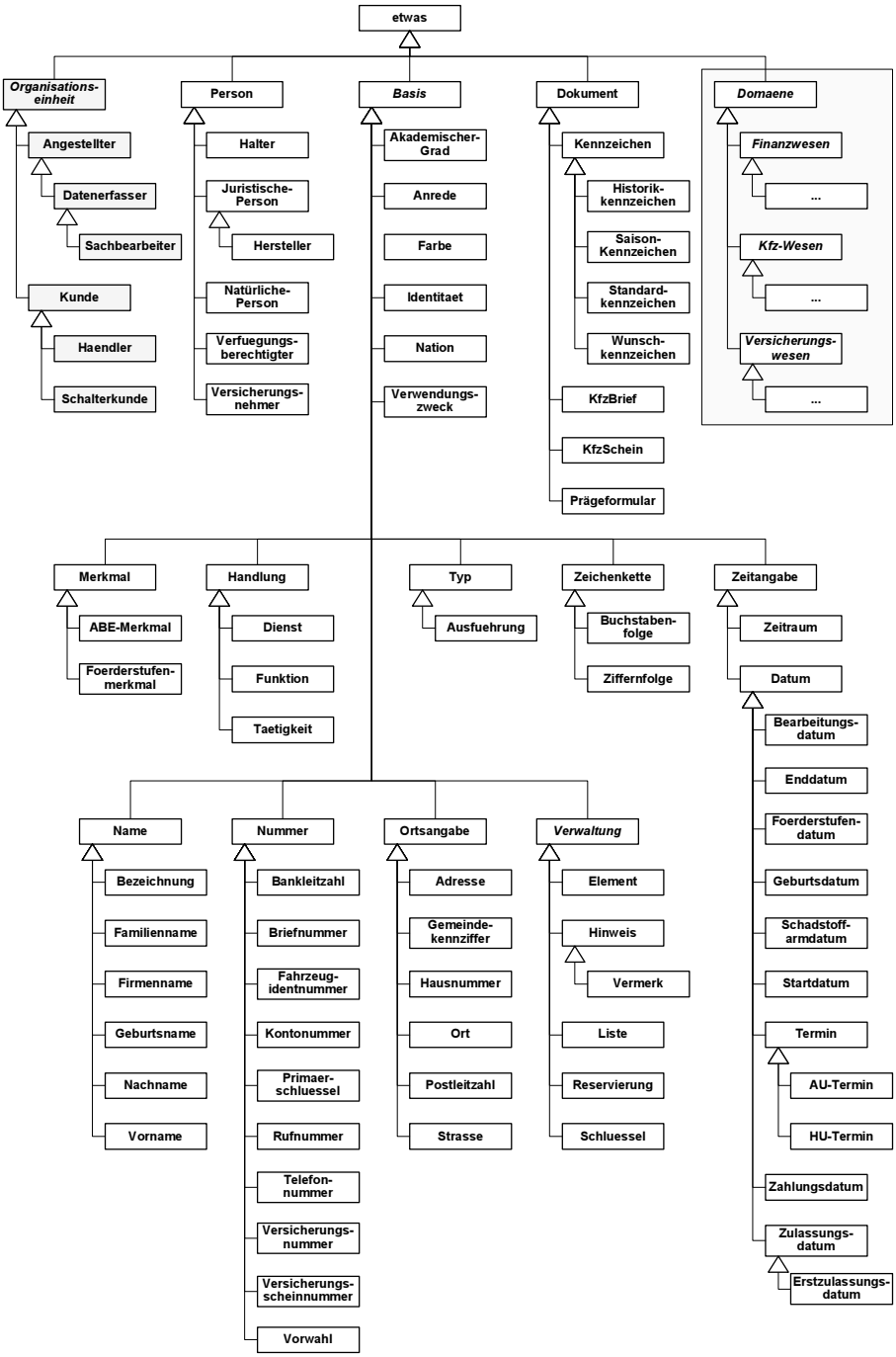


Abbildung C.2: Terminologie (Konzepte)

In Abbildung C.2 haben wir die Konzepte, die in unserem terminologischen Modell gleichzeitig auch Organisationseinheiten sind, grau unterlegt. Die Darstellung der für die betrachtete Kfz-Domäne spezifischen Begriffe, die keine Generalisierungs-/Spezialisierungsbeziehungen zu domänenunabhängigen Begriffen aufweisen (oben rechts in der Abbildung), haben wir aus Gründen der Übersichtlichkeit in Abbildung C.3 ausgelagert.

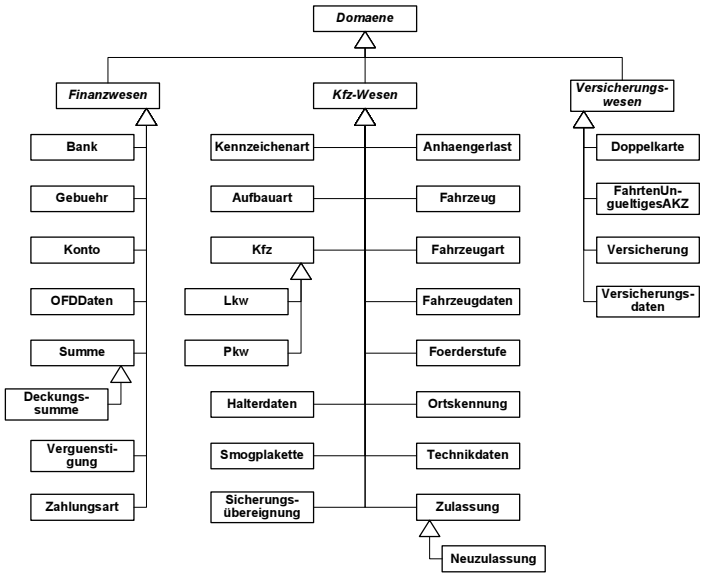


Abbildung C.3: Terminologie (domänenspezifische Konzepte)

Da Ganzes/Teile-Beziehungen zwischen Prädikatoren für den Suchprozess irrelevant sind, haben wir uns bei den vorangegangenen Abbildungen zugunsten einer höheren Übersichtlichkeit auf die Darstellung der Generalisierungs-/Spezialisierungsbeziehungen beschränkt.

C.2 Geschäftsprozessmodelle

Die Geschäftsprozessmodelle aus dem Bereich des Kfz-Zulassungswesens, die für die Evaluation zur Verfügung standen, orientieren sich in weiten Teilen an dem Aufbau von Bildschirmmasken, die ein auf der Grundlage von Komponenten zu entwickelndes Zielsystem nach Möglichkeit anbieten sollte. Der Zusammenhang zwischen diesen Geschäftsprozessmodellen, der aufgrund von Hierarchisierungsbeziehungen zwischen den einzelnen Prozessmodellen besteht, ist in Abbildung C.4 graphisch dargestellt.

Im Folgenden spezifizieren wir die aus den verfügbaren Aktivitätsdiagrammen abgeleiteten linearen Prozessmodelle durch ihre prozessalgebraische Repräsentation. Verweise auf untergeordnete Prozesse kennzeichnen wir dabei durch das Symbol → und die Bezeichnung des Prozesses. Bezüglich der in Abbildung C.4 dargestellten Prozesshierarchie bewegen wir uns bei der Vorstellung der Prozessmodelle von links nach rechts und von unten nach oben.

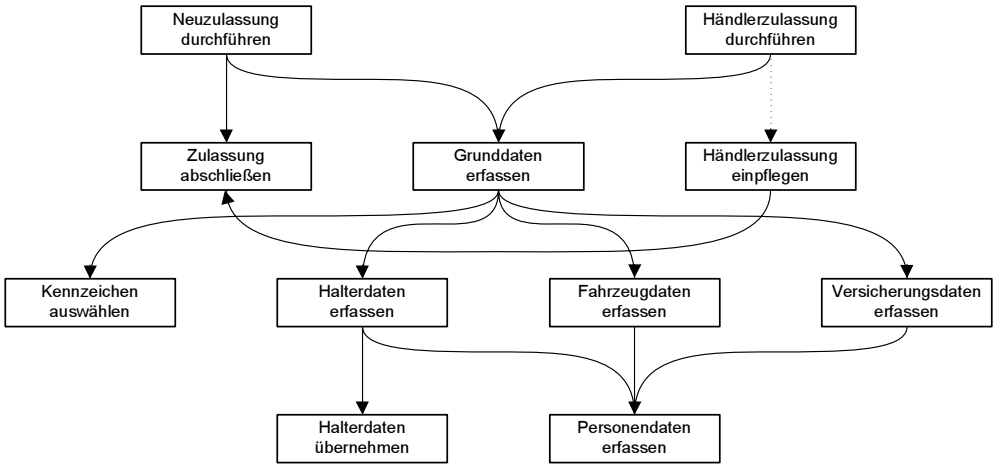


Abbildung C.4: Zusammenhang zwischen Geschäftsprozessmodellen

Halterdaten übernehmen Der Geschäftsprozess „Halterdaten übernehmen“ beschreibt die Übernahme bestehender Halterdaten, z. B. bei Zulassung eines Zweitwagens.

Halterdaten übernehmen
= $\mathcal{S}$($\mathcal{A}$($\mathcal{S}$(Datenerfasser ! erfassen Ortskennung,
Datenerfasser ! erfassen Buchstabenfolge,
Datenerfasser ! erfassen Ziffernfolge),
$\mathcal{S}$(Sachbearbeiter ! erfassen Name,
Sachbearbeiter ! erfassen Geburtsdatum))
Datenerfasser ! suchen Halterdaten,
$\mathcal{A}$(Sachbearbeiter ! aendern Halterdaten,
ϵ,
Sachbearbeiter ! speichern Halterdaten))

Personendaten erfassen Der Geschäftsprozess „Personendaten erfassen“ dient der Erfassung der Daten von Personen, die noch nicht als Halter im System registriert sind oder aber für eine Sicherungsübereignung eines Fahrzeugs benötigt werden.

Personendaten erfassen
= $\mathcal{A}$($\mathcal{S}$(Datenerfasser ! erfassen Name,
Datenerfasser ! erfassen Adresse,
Datenerfasser ! erfassen Geburtsdatum),
$\mathcal{S}$(Datenerfasser ! erfassen Firmenname,
Datenerfasser ! erfassen Adresse))

Kennzeichen auswählen Über den Geschäftsprozess „Kennzeichen auswählen“ kann ein Kennzeichen für eine Kfz-Zulassung ausgewählt werden. Dabei kann zwischen dem als nächstes verfügbaren Kennzeichen, einem Wunschkennzeichen oder einem zuvor reservierten Kennzeichen gewählt werden.

Kennzeichen auswählen
= $\mathcal{S}$(Datenerfasser ! erfassen Kennzeichenart,
$\mathcal{A}$($\mathcal{S}$($\mathcal{A}$(Datenerfasser ! suchen Wunschkennzeichen,
ϵ),
Datenerfasser ! auswaehlen Kennzeichen),
$\mathcal{S}$($\mathcal{A}$(ϵ,
$\mathcal{A}$($\mathcal{S}$(Sachbearbeiter ! erfassen Name,
Sachbearbeiter ! erfassen Geburtsdatum),
$\mathcal{S}$(Datenerfasser ! erfassen Ortskennung,
Datenerfasser ! erfassen Buchstabenfolge,
Datenerfasser ! erfassen Ziffernfolge))),
$\mathcal{S}$(Datenerfasser ! suchen Reservierung,
Datenerfasser ! anzeigen Reservierung,
Datenerfasser ! auswaehlen Kennzeichen,
$\mathcal{A}$(ϵ,
$\mathcal{A}$(ϵ,
Datenerfasser ! loeschen Reservierung))))))

Halterdaten erfassen Der Geschäftsprozess „Halterdaten erfassen“ dient der Erfassung der für die Zulassung eines Fahrzeugs erforderlichen Daten zur Beschreibung des Halters. Sollten die erforderlichen Halterdaten bereits im System vorhanden sein, können diese übernommen und ggf. geändert werden, ansonsten sind die Daten neu zu erfassen.

Halterdaten erfassen
= $\mathcal{A}$($\rightarrow$ Halterdaten übernehmen,
$\mathcal{S}$($\rightarrow$ Personendaten erfassen,
$\mathcal{A}$(ϵ,
Datenerfasser ! bearbeiten Hinweis),
$\mathcal{A}$(ϵ,
Datenerfasser ! erfassen Name),
$\mathcal{A}$(Datenerfasser ! vorbelegen Berufsschluessel,
Datenerfasser ! erfassen Berufsschluessel),
$\mathcal{A}$(ϵ,
Datenerfasser ! erfassen Ortsangabe),
$\mathcal{A}$(ϵ,
$\mathcal{S}$(Datenerfasser ! erfassen Bankleitzahl,
Datenerfasser ! erfassen Kontonummer)),
$\mathcal{A}$(Sachbearbeiter ! speichern Halterdaten,
Datenerfasser ! zwischenspeichern Halterdaten)))

Fahrzeugdaten erfassen Mit dem Geschäftsprozess „Fahrzeugdaten erfassen“ werden die für die Zulassung eines Fahrzeugs relevanten Daten aufgenommen. Soll eine Sicherungsübereignung eingetragen werden, werden die Daten der Person erfasst, an die das Fahrzeug übereignet wird.

Fahrzeugdaten erfassen
= $\mathcal{S}$(Datenerfasser ! erfassen Briefnummer,
Datenerfasser ! erfassen Sicherungsuebereignung,
$\mathcal{A}$(ϵ,
→ Personendaten erfassen),
Datenerfasser ! erfassen Hersteller,
Datenerfasser ! erfassen Typ,
Datenerfasser ! erfassen Ausfuehrung,
Datenerfasser ! erfassen Pruefziffer,
$\mathcal{A}$($\mathcal{S}$(Datenerfasser ! erfassen Fahrzeugart,
Datenerfasser ! erfassen Aufbauart),
Datenerfasser ! vorbelegen Fahrzeugdaten),
Datenerfasser ! erfassen ABE-Merkmal,
$\mathcal{A}$(ϵ,
Datenerfasser ! bearbeiten Technikdaten),
Datenerfasser ! erfassen Fahrzeugidentnummer,
Datenerfasser ! erfassen Pruefziffer,
Datenerfasser ! erfassen Farbe,
$\mathcal{A}$(ϵ,
Datenerfasser ! erfassen Erstzulassungsdatum),
Datenerfasser ! erfassen Zulassungsdatum,
Datenerfasser ! erfassen HU-Termin,
Datenerfasser ! erfassen AU-Termin,
Datenerfasser ! erfassen Verwendungszweck,
Datenerfasser ! erfassen OFDDaten,
$\mathcal{A}$(Datenerfasser ! speichern Fahrzeugdaten,
Datenerfasser ! zwischenspeichern Fahrzeugdaten))

Versicherungsdaten erfassen Der Geschäftsprozess „Versicherungsdaten erfassen“ dient der Aufnahme der Angaben, mit denen der Versicherungsschutz des zuzulassenden Fahrzeugs sichergestellt wird.

Versicherungsdaten erfassen
= $\mathcal{S}$(Datenerfasser ! erfassen Nation,
$\mathcal{A}$($\mathcal{S}$(Datenerfasser ! erfassen Versicherungsnummer,
Datenerfasser ! erfassen Versicherungsscheinnummer,
Datenerfasser ! erfassen Doppelkarte,
Datenerfasser ! erfassen Versicherung,
Datenerfasser ! erfassen Deckungssumme,
Datenerfasser ! erfassen Vermerk),
ϵ),
Datenerfasser ! erfassen FahrtenUngueltigesAkz,
$\mathcal{A}$(ϵ,
→ Personendaten erfassen), ...

```
...     A(ε,
          Datenerfasser ! erfassen Zeitraum),
        Datenerfasser ! erfassen Startdatum,
        Datenerfasser ! erfassen Enddatum,
        A(Datenerfasser ! speichern Versicherungsdaten,
          Datenerfasser ! zwischenspeichern Versicherungsdaten))
```

Zulassung abschließen Mit dem Geschäftsprozess „Zulassung abschließen“ wird ein Zulassungsvorgang – sofern die erhobenen Gebühren entrichtet werden – abgeschlossen und die erforderlichen Dokumente gedruckt, anderenfalls wird der Vorgang abgebrochen.

```
Zulassung abschließen
  =  S(Sachbearbeiter ! berechnen Gebuehr,
       A(Sachbearbeiter ! abbrechen Zulassung,
         S(Sachbearbeiter ! speichern Zulassung,
           Sachbearbeiter ! fordern Gebuehr,
           Schalterkunde ! bezahlen Gebuehr,
           A(ε,
             Sachbearbeiter ! drucken Dokument))))
```

Grunddaten erfassen Der Geschäftsprozess „Grunddaten erfassen“ dient der Erfassung aller Daten, die sowohl für eine Zulassung bei der Zulassungsstelle als auch durch einen Kfz-Händler benötigt werden.

```
Grunddaten erfassen
  =  S(→ Kennzeichen auswaehlen,
       → Halterdaten erfassen,
       A(Datenerfasser ! drucken Praegeformular,
         ε),
       → Fahrzeugdaten erfassen,
       → Versicherungsdaten erfassen)
```

Händlerzulassung einpflegen Über den Geschäftsprozess „Händlerzulassung einpflegen“ werden Zulassungsvorgänge, die von einem Kfz-Händler initiiert wurden, bei der Zulassungsstelle abgeschlossen.

```
Händlerzulassung einpflegen
  =  S(Schalterkunde ! identifizieren Person,
       A(S(Sachbearbeiter ! auswaehlen Haendler,
           Sachbearbeiter ! suchen Zulassung,
           A(S(Sachbearbeiter ! aendern Zulassung,
               → Zulassung abschließen),
             ε)),
         ε))
```

Neuzulassung durchführen Mit dem Geschäftsprozess „Neuzulassung durchführen“ erfolgt die Neuzulassung eines Fahrzeugs bei der Zulassungsstelle.

Neuzulassung durchführen
= $\mathcal{S}$(Kunde ! belegen Identitaet,
Kunde ! bereitstellen KfzBrief,
Kunde ! bereitstellen Doppelkarte,
→ Grunddaten erfassen,
→ Zulassung abschließen)

Händlerzulassung durchführen Der Geschäftsprozess „Händlerzulassung durchführen“ ermöglicht es einem Kfz-Händler, eine Neuzulassung für den Besuch bei der Zulassungsstelle vorzubereiten. Diese muss dann später über den Prozess „Händlerzulassung einpflegen“ bei der Zulassungsstelle abgeschlossen werden.

Händlerzulassung durchführen
= $\mathcal{S}$(Haendler ! authentifizieren Person,
$\mathcal{A}$(ϵ,
$\mathcal{S}$(→ Grunddaten erfassen,
Haendler ! zwischenspeichern Neuzulassung)))

C.3 Komponentenbeschreibungen

Für die Evaluation der geschäftsprozessorientierten Komponentensuche standen insgesamt elf EJB-Komponenten zur Verfügung, unter denen sich neben neun Entity Beans je eine zustandslose und eine zustandsbehaftete Session Bean befanden. In diesem Abschnitt dokumentieren wir diese Komponenten durch ihre CDL-Beschreibungen. Dabei weist die zustandsbehaftete Session Bean als Ausprägung einer typischen Prozesskomponente einen Lebenszyklus auf, während dies bei der zustandslosen Session Bean aufgrund ihres zustandslosen Charakters nicht der Fall ist. Die Verfügbarkeit der Operationen von Entity Beans haben wir nur dann durch ihre Verhaltensbeschreibungen zustandsabhängig eingeschränkt, wenn die Eigenschaften (Attribute) der durch die Komponente persistent verwalteten Geschäftsobjekte auf einen Lebenszyklus der Objekte hindeuten. Reine Datenhaltungskomponenten ohne solche Eigenschaften weisen keine nennenswerten Einschränkungen bzgl. der Verfügbarkeit ihrer Operationen auf.

Zur Typisierung der Slots der nachfolgend vorgestellten Komponenten haben wir so genannte *Standardkontrakte* verwendet. Ein Standardkontrakt ist direkt aus der Beschreibung einer Komponente abgeleitet, die diesen Kontrakt dann auch unterstützt. Er trägt den Namen dieser Komponente, ergänzt um ein „`C`“, und geht aus der Komponentenbeschreibung durch syntaktische Ersetzungen der Schlüsselwörter sowie der Argumente von `new`- und `existing`-Operatoren hervor.

Da Entity Beans oft eine große Anzahl von Eigenschaften definieren, die einen einfachen Datentyp haben und daher den Suchprozess nicht durch die Referenzierung neuer bzw. existierender Objekte beeinflussen, führen wir eine abkürzende Schreibweise für Operationen ein, die den Zugriff auf solche Eigenschaften ermöglichen. Dabei steht ein Ausdruck der Form

```
property <propertyname>: <propertytype>
  [ Handlungsträger γ <GetPrädikat> <DirektesObjekt> ]
  [ Handlungsträger γ <SetPrädikat> <DirektesObjekt> ]
```

für den CDL-Ausdruck

```
get<Propertyname>(out <propertyname>: <propertytype>)
  [ Handlungsträger γ <GetPrädikat> <DirektesObjekt> ] {
  return (*);
};
set<Propertyname>(in <propertyname>: <propertytype>)
  [ Handlungsträger γ <SetPrädikat> <DirektesObjekt> ] {
  return;
};
```

Wie verzichten darauf, Operationen zur Pflege von Beziehungen zwischen Geschäftsobjekten eine fachliche Semantik zuzuordnen, da diese einen primär technischen Charakter haben und nicht direkt zur Ausführung von Geschäftsprozessen beitragen.

Dienst_KennzeichenBean Das Listing C.1 zeigt die CDL-Beschreibung der zustandslosen Session Bean `Dienst_KennzeichenBean`, die Funktionalität für die Suche nach einem freien Kennzeichen realisiert.

```
Component Dienst_KennzeichenBean {

  slot KennzeichenBeanC kennzeichenComp;

  class Dienst_Kennzeichen {
    sucheNaechstesKennzeichen(in sArt: String, in nSuchmuster: String,
                              out kennzeichen: Kennzeichen)
      [ Handlungsträger γ suchen Kennzeichen ] {
      existing(KennzeichenBeanC.Kennzeichen kennzeichen);
      return(kennzeichen);
    };
    holeNaechstesFreiesAKZ(out freiesAKZ: Kennzeichen)
      [ Handlungsträger γ ermitteln Kennzeichen ]
      [ Handlungsträger γ bereitstellen Kennzeichen ] {
      existing(KennzeichenBeanC.Kennzeichen kennzeichen);
      return(kennzeichen);
    };

    created()     =   initialized();
    initialized() =   (sucheNaechstesKennzeichen?
                          (sArt, nSuchmuster, kennzeichen)
                       + holeNaechstesFreiesAKZ?(freiesAKZ)).initialized();
  };

  create(out dienst: Dienst_Kennzeichen)
    [ Handlungsträger γ anlegen Dienst ] {
    new (Dienst_KennzeichenBean.Dienst_Kennzeichen dienst);
    return (dienst);
  };

  created()     = initialized();
  initialized() = create?(dienst).initialized();
};
```

Listing C.1: CDL-Beschreibung der Session Bean `Dienst_KennzeichenBean`

Manager_ZulassungBean Die zustandsbehaftete Session Bean `Manager_ZulassungBean`, deren CDL-Beschreibung in Listing C.2 wiedergegeben ist, stellt einen Dienst für die Durchführung von Kfz-Neuzulassungen zur Verfügung.

```
Component Manager_ZulassungBean {

  slot KennzeichenBeanC kennzeichenComp;
  slot HalterBeanC halterComp;
  slot FahrzeugdatenBeanC fahrzeugdatenComp;
  slot VersicherungsdatenBeanC versicherungsdatenComp;

  class Manager_Zulassung {
    property user: String
      [ Handlungsträger γ ermitteln Datenerfasser ]
      [ Handlungsträger γ festlegen Datenerfasser ]

    starteNeuzulassung(in user: String, in date: Object)
      [ Handlungsträger γ anlegen Neuzulassung ]
      [ Handlungsträger γ beginnen Neuzulassung ] {
      return;
    };
    getDate(out date: Object)
      [ Handlungsträger γ ermitteln Zulassungsdatum ] {
      return (*);
    };
    naechstesFreieKZ(in type: int, out kennzeichen: Kennzeichen)
      [ Handlungsträger γ ermitteln Kennzeichen ] {
      existing(KennzeichenBeanC.Kennzeichen kennzeichen);
      return(kennzeichen);
    };
    erfasseHalterdatenByVO(in personVO: Person, in adresseVO: Adresse,
                           in halterVO: Halter)
      [ Handlungsträger γ erfassen Halterdaten ] {
      return;
    };
    speichere(out success: boolean)
      [ Handlungsträger γ abschliessen Zulassung ]
      [ Handlungsträger γ speichern Zulassung ]
      [ Handlungsträger γ speichern Halterdaten ]
      [ Handlungsträger γ speichern Fahrzeugdaten ]
      [ Handlungsträger γ speichern Versicherungsdaten ] {
      return (*);
    };
    setAkz(in akz: Kennzeichen)
      [ Handlungsträger γ festlegen Kennzeichen ] {
      return;
    };
    erfasseHalterdaten(in sName: String, in sVorname: String,
                       in sAnrede: String, in einBerufsschluessel: String,
                       out success: boolean)
      [ Handlungsträger γ erfassen Halterdaten ]
      [ Handlungsträger γ erfassen Halter ]
      [ Handlungsträger γ erfassen Vorname ]
      [ Handlungsträger γ erfassen Nachname ]
      [ Handlungsträger γ erfassen Anrede ]
      [ Handlungsträger γ erfassen Berufsschluessel ] {
      return (*);
    };
    erfasseFahrzeugdatenByVO(in data: Fahrzeugdaten, in ofdDatum: Object,
                             out success: boolean)
      [ Handlungsträger γ erfassen Fahrzeugdaten ] {
      return (*);
    };
```

```
erfasseVersicherungsdaten(in sNation: String, in sVersicherungsNR: String,
                          in sScheinNr: String, in sinterneVermerke: String,
                          in sDeckungssumme: String,
                          in bUngestempeltesAKZ: boolean,
                          in einVersicherungsbeginn: Object,
                          in einVersicherungsEnde: Object,
                          out success: boolean)
  [ Handlungsträger γ erfassen Versicherungsdaten ]
  [ Handlungsträger γ erfassen Versicherungsnummer ]
  [ Handlungsträger γ erfassen Versicherungsscheinnummer ]
  [ Handlungsträger γ erfassen Vermerk ]
  [ Handlungsträger γ erfassen Deckungssumme ]
  [ Handlungsträger γ erfassen FahrtenUngueltigesAkz ]
  [ Handlungsträger γ erfassen Startdatum ]
  [ Handlungsträger γ erfassen Enddatum ]
  [ Handlungsträger γ erfassen Nation ] {
  return (*);
};
erfasseFahrzeugdaten(in sBriefNr: String, in nSicherung: int,
                     in nHersteller: long, in nTyp: long,
                     in nAusfuehrung: long, in nPruefziffer: int,
                     in sFahrzeug: String, in sAufbauart: String,
                     in nABEMerkmal: int, in FahrzeugID_PZ: int,
                     in nFarbe1: int, in nFarbe2: int,
                     in einZulassungsDatum: Object,
                     in einNaechstesHU_Datum: Object,
                     in einNaechstesAU_Datum: Object,
                     out success: boolean)
  [ Handlungsträger γ erfassen Fahrzeugdaten ]
  [ Handlungsträger γ erfassen Briefnummer ]
  [ Handlungsträger γ erfassen Sicherungsuebereignung ]
  [ Handlungsträger γ erfassen Hersteller ]
  [ Handlungsträger γ erfassen Typ ]
  [ Handlungsträger γ erfassen Ausfuehrung ]
  [ Handlungsträger γ erfassen Pruefziffer ]
  [ Handlungsträger γ erfassen Bezeichnung ]
  [ Handlungsträger γ erfassen Aufbauart ]
  [ Handlungsträger γ erfassen ABE-Merkmal ]
  [ Handlungsträger γ erfassen Pruefziffer ]
  [ Handlungsträger γ erfassen Farbe ]
  [ Handlungsträger γ erfassen Zulassungsdatum ]
  [ Handlungsträger γ erfassen HU-Termin ]
  [ Handlungsträger γ erfassen AU-Termin ] {
  return (*);
};

created()     =   initialized();
initialized() =   (getDate?(date) + setUser?(sUser)
                  + getUser?(user)).initialized()
                + starteNeuzulassung?(user, date).gestartet();
gestartet()   =   speichere?(success).gespeichert()
                + (getUser?(user)
                  + erfasseVersicherungsdaten?(sNation, sVersicherungsNR,
                      sScheinNr, sinterneVermerke, sDeckungssumme,
                      bUngestempeltesAKZ, einVersicherungsbeginn,
                      einVersicherungsEnde, success)
                  + setAkz?(akz)
                  + erfasseFahrzeugdatenByVO?(data, ofdDatum, success)
                  + erfasseHalterdaten?(sName, sVorname, sAnrede,
                      einBerufsschluessel, success)
                  + getDate?(date)
                  + erfasseHalterdaten?(sName, sVorname, sAnrede,
                      einBerufsschluessel, success)
```

```
                        + erfasseFahrzeugdaten?(sBriefNr, nSicherung,
                            nHersteller, nTyp, nAusfuehrung, nPruefziffer,
                            sFahrzeug, sAufbauart, nABEMerkmal, FahrzeugID_PZ,
                            nFarbe1, nFarbe2, einZulassungsDatum,
                            einNaechstesHU_Datum, success)
                        + naechstesFreieKZ?(type, kennzeichen)
                        + erfasseHalterdatenByVO?(personVO, adresseVO, halterVO))
                        .gestartet();
    gespeichert() =     (getUser?(user) + getDate?(date)).gespeichert();
  };

  create(out dienst: Manager_Zulassung)
    [ Handlungsträger γ anlegen Dienst ] {
    new (Manager_ZulassungBean.Manager_Zulassung dienst);
    return (dienst);
  };

  created()     = initialized();
  initialized() = create?(dienst).initialized();
};
```

Listing C.2: CDL-Beschreibung der Session Bean `Manager_ZulassungBean`

AdresseBean Mit der Entity Bean `AdresseBean`, die in Listing C.3 dokumentiert ist, lassen sich die Adressen von Personen bzw. die Standorte von Kraftfahrzeugen verwalten.

```
Component AdresseBean {

  slot KfzBeanC kfzComp;
  slot PersonBeanC personComp;

  class Adresse {
    property strasse: String
      [ Handlungsträger γ ermitteln Strasse ]
      [ Handlungsträger γ festlegen Strasse ]
    property hausNr: int
      [ Handlungsträger γ ermitteln Hausnummer ]
      [ Handlungsträger γ festlegen Hausnummer ]
    property hausBuchstabe: String
      [ Handlungsträger γ ermitteln Hausnummer ]
      [ Handlungsträger γ festlegen Hausnummer ]
    property gkz: String
      [ Handlungsträger γ ermitteln Gemeindekennziffer ]
      [ Handlungsträger γ festlegen Gemeindekennziffer ]
    property plz: String
      [ Handlungsträger γ ermitteln Postleitzahl ]
      [ Handlungsträger γ festlegen Postleitzahl ]
    property ort: String
      [ Handlungsträger γ ermitteln Ort ]
      [ Handlungsträger γ festlegen Ort ]

    getKfz(out kfz: Kfz) {
      existing (KfzBeanC.Kfz kfz);
      return (kfz);
    };
    setKfz(in kfz: Kfz) {
      return;
    };
```

```
    getPerson(out person: Person) {
      existing (PersonBeanC.Person person);
      return (person);
    };
    setPerson(in person: Person) {
      return;
    };

    created()     =   initialized();
    initialized() =   (getStrasse?(strasse) + getHausNr?(hausNr) + setKfz?(kfz)
                      + setHausBuchstabe?(hausBuchstabe) + setGkz?(gkz)
                      + getPlz?(plz) + setHausNr?(hausNr)
                      + getHausBuchstabe?(hausBuchstabe) + getGkz?(gkz)
                      + getPerson?(person) + setStrasse?(strasse) + getKfz?(kfz)
                      + setOrt?(ort) + setPlz?(plz) + getOrt?(ort)
                      + setPerson?(person)).initialized();
  };

  create(out adresse: Adresse)
    [ Handlungsträger γ anlegen Adresse ] {
    new (AdresseBean.Adresse adresse);
    return (adresse);
  };
  findByPrimaryKey(in key: String, out result: Set<AdresseBean.Adresse>)
    [ Handlungsträger γ suchen Adresse ] {
    existing(Set<AdresseBean.Adresse> result);
    return(result);
  };
  remove(in key: String)
    [ Handlungsträger γ loeschen Adresse ] {
    return;
  };

  created()     =   initialized();
  initialized() =   (findByPrimaryKey?(key, result) + create?(adresse)
                    + remove?(key)).initialized();
};
```

Listing C.3: CDL-Beschreibung der Entity Bean `AdresseBean`

FahrzeugdatenBean Die für die Zulassung eines Kraftfahrzeugs erforderlichen Angaben zum Fahrzeug können mit der Entity Bean `FahrzeugdatenBean` verwaltet werden. Ihre CDL-Beschreibung ist in Listing C.4 dargestellt.

```
Component FahrzeugdatenBean {

  slot OFDDatenBeanC ofdDatenComp;
  slot TechnikdatenBeanC technikdatenComp;
  slot KfzBeanC kfzComp;

  class Fahrzeugdaten {
    property briefnummer: String
      [ Handlungsträger γ ermitteln Briefnummer ]
      [ Handlungsträger γ festlegen Briefnummer ]
    property sicherung: int
      [ Handlungsträger γ ermitteln Sicherungsuebereignung ]
      [ Handlungsträger γ festlegen Sicherungsuebereignung ]
    property hersteller: long
      [ Handlungsträger γ ermitteln Hersteller ]
      [ Handlungsträger γ festlegen Hersteller ]
```

```
property typ: long
  [ Handlungsträger γ ermitteln Typ ]
  [ Handlungsträger γ festlegen Typ ]
property ausfuehrung: long
  [ Handlungsträger γ ermitteln Ausfuehrung ]
  [ Handlungsträger γ festlegen Ausfuehrung ]
property pruefziffer: int
  [ Handlungsträger γ ermitteln Pruefziffer ]
  [ Handlungsträger γ festlegen Pruefziffer ]
property abeMerkmal: int
  [ Handlungsträger γ ermitteln ABE-Merkmal ]
  [ Handlungsträger γ festlegen ABE-Merkmal ]
property farbe1: int
  [ Handlungsträger γ ermitteln Farbe ]
  [ Handlungsträger γ festlegen Farbe ]
property farbe2: int
  [ Handlungsträger γ ermitteln Farbe ]
  [ Handlungsträger γ festlegen Farbe ]
property ezDatum: Object
  [ Handlungsträger γ ermitteln Erstzulassungsdatum ]
  [ Handlungsträger γ festlegen Erstzulassungsdatum ]
property zulassungsdatum: Object
  [ Handlungsträger γ ermitteln Zulassungsdatum ]
  [ Handlungsträger γ festlegen Zulassungsdatum ]
property naechsteHU: Object
  [ Handlungsträger γ ermitteln HU-Termin ]
  [ Handlungsträger γ festlegen HU-Termin ]
property naechsteAU: Object
  [ Handlungsträger γ ermitteln AU-Termin ]
  [ Handlungsträger γ festlegen AU-Termin ]
property verwendungszweck: int
  [ Handlungsträger γ ermitteln Verwendungszweck ]
  [ Handlungsträger γ festlegen Verwendungszweck ]
property fahrzeug: String
  [ Handlungsträger γ ermitteln Bezeichnung ]
  [ Handlungsträger γ festlegen Bezeichnung ]
property aufbauart: String
  [ Handlungsträger γ ermitteln Aufbauart ]
  [ Handlungsträger γ festlegen Aufbauart ]
property fahrzeugID: String
  [ Handlungsträger γ ermitteln Fahrzeugidentnummer ]
  [ Handlungsträger γ festlegen Fahrzeugidentnummer ]
property fahrzeugID_PZ: int
  [ Handlungsträger γ ermitteln Pruefziffer ]
  [ Handlungsträger γ festlegen Pruefziffer ]

getOfdDaten(out ofdDaten: OFDDaten) {
  existing (OFDDatenBeanC.OFDDaten oFDDaten);
  return (oFDDaten);
};
setOfdDaten(in ofdDaten: OFDDaten) {
  return;
};
getKfz(out kfz: Kfz) {
  existing (KfzBeanC.Kfz kfz);
  return (kfz);
};
setKfz(in kfz: Kfz) {
  return;
};

created()     = initialized();
initialized() = (setKfz?(kfz) + getOfdDaten?(ofdDaten)
                 + setSicherung?(sicherung) + getFarbe2?(farbe2)
```

```
                    + setAusfuehrung?(ausfuehrung) + setNaechsteAU?(naechsteAU)
                    + getAufbauart?(aufbauart) + setAufbauart?(aufbauart)
                    + getAbeMerkmal?(abeMerkmal) + setNaechsteHU?(naechsteHU)
                    + setTechnik?(technik) + getFahrzeugID_PZ?(fahrzeugID_PZ)
                    + setTyp?(typ) + setOfdDaten?(ofdDaten)
                    + setHersteller?(hersteller) + getHersteller?(hersteller)
                    + setEzDatum?(ezDatum) + getPruefziffer?(pruefziffer)
                    + setFarbe2?(farbe2) + setZulassungsdatum?(zulassungsdatum)
                    + getFahrzeugID?(fahrzeugID) + setFarbe1?(farbe1)
                    + getKfz?(kfz) + setFahrzeugID_PZ?(fahrzeugID_PZ)
                    + setPruefziffer?(pruefziffer) + getSicherung?(sicherung)
                    + getFahrzeug?(fahrzeug) + getEzDatum?(ezDatum)
                    + getVerwendungszweck?(verwendungszweck)
                    + setAbeMerkmal?(abeMerkmal) + getAusfuehrung?(ausfuehrung)
                    + getNaechsteHU?(naechsteHU) + setFahrzeugID?(fahrzeugID)
                    + getZulassungsdatum?(zulassungsdatum)
                    + setFahrzeug?(fahrzeug) + getNaechsteAU?(naechsteAU)
                    + setVerwendungszweck?(verwendungszweck)
                    + getTechnik?(technik) + getFarbe1?(farbe1)
                    + setBriefnummer?(briefnummer) + getBriefnummer?(briefnummer)
                    + getTyp?(typ)).initialized();
  };

  create(out fahrzeugdaten: Fahrzeugdaten)
    [ Handlungsträger γ anlegen Fahrzeugdaten ] {
    new (FahrzeugdatenBean.Fahrzeugdaten fahrzeugdaten);
    return (fahrzeugdaten);
  };
  findByPrimaryKey(in key: String,
                   out result: Set<FahrzeugdatenBean.Fahrzeugdaten>)
    [ Handlungsträger γ suchen Fahrzeugdaten ] {
    existing(Set<FahrzeugdatenBean.Fahrzeugdaten> result);
    return(result);
  };
  remove(in key: String)
    [ Handlungsträger γ loeschen Fahrzeugdaten ] {
    return;
  };

  created()     = initialized();
  initialized() = (findByPrimaryKey?(key, result)
                  + create?(fahrzeugdaten) + remove?(key)).initialized();
};
```

Listing C.4: CDL-Beschreibung der Entity Bean `FahrzeugdatenBean`

HalterBean Halter eines Kraftfahrzeugs lassen sich mit Hilfe der in Listing C.5 beschriebenen Entity Bean `HalterBean` persistent verwalten.

```
Component HalterBean {

  slot PersonBeanC personComp;
  slot KfzBeanC kfzComp;

  class Halter {
    property hinweiseUeberlangerName: String
      [ Handlungsträger γ ermitteln Hinweis ]
      [ Handlungsträger γ festlegen Hinweis ]
    property zweiterHalter: boolean
      [ Handlungsträger γ ermitteln Halter ]
      [ Handlungsträger γ festlegen Halter ]
```

```
    property id: String
      [ Handlungsträger γ ermitteln Primaerschluessel ]
      [ Handlungsträger γ festlegen Primaerschluessel ]
    property berufsschluessel: String
      [ Handlungsträger γ ermitteln Berufsschluessel ]
      [ Handlungsträger γ festlegen Berufsschluessel ]

    getPerson(out person: Person) {
      existing (PersonBeanC.Person person);
      return (person);
    };
    setPerson(in person: Person) {
      return;
    };
    getKfz(out kfz: Kfz) {
      existing (KfzBeanC.Kfz kfz);
      return (kfz);
    };
    setKfz(in kfz: Kfz) {
      return;
    };

    created()        =   initialized();
    initialized()    =   setId?(id).id_gesetzt();
    id_gesetzt()     =   getId?(id).id_gesetzt()
                       + setPerson?(person).person_gesetzt();
    person_gesetzt() =   setKfz?(kfz).kfz_zugeordnet()
                       + (getPerson?(person) + getId?(id)
                         + setBerufsschluessel?(berufsschluessel)
                         + getHinweiseUeberlangerName?(hinweiseUeberlangerName)
                         + setHinweiseUeberlangerName?(hinweiseUeberlangerName)
                         + getBerufsschluessel?(berufsschluessel))
                         .person_gesetzt();
    kfz_zugeordnet() =   (getId?(id) + getKfz?(kfz) + getPerson?(person)
                         + setZweiterHalter?(zweiterHalter)
                         + getZweiterHalter?(zweiterHalter)).kfz_zugeordnet()
                       + setKfz?(kfz).person_gesetzt();
  };

  create(out halter: Halter)
    [ Handlungsträger γ anlegen Halter ]
    [ Handlungsträger γ anlegen Halterdaten ] {
    new (HalterBean.Halter halter);
    return (halter);
  };
  findByPrimaryKey(in key: String, out result: Set<HalterBean.Halter>)
    [ Handlungsträger γ suchen Halter ]
    [ Handlungsträger γ suchen Halterdaten ] {
    existing(Set<HalterBean.Halter> result);
    return(result);
  };
  remove(in key: String)
    [ Handlungsträger γ loeschen Halter ]
    [ Handlungsträger γ loeschen Halterdaten ] {
    return;
  };

  created()     =   initialized();
  initialized() =   (create?(halter) + findByPrimaryKey?(key, result)
                    + remove?(key)).initialized();
};
```

Listing C.5: CDL-Beschreibung der Entity Bean `HalterBean`

KennzeichenBean Die Entity Bean `KennzeichenBean`, deren CDL-Beschreibung in Listing C.6 dargestellt ist, dient der Verwaltung von amtlichen Kennzeichen. Ein Kennzeichen-Geschäftsobjekt kann dabei in einem der Zustände `initialized` (verfügbar), `reserviert` oder `vergeben` sein.

```
Component KennzeichenBean {

  slot ReservierungBeanC reservierungComp;
  slot KfzBeanC kfzComp;

  class Kennzeichen {
    property id: String
      [ Handlungsträger γ ermitteln Primaerschluessel ]
      [ Handlungsträger γ festlegen Primaerschluessel ]
    property type: int
      [ Handlungsträger γ ermitteln Kennzeichenart ]
      [ Handlungsträger γ festlegen Kennzeichenart ]
    property bisDatum: Object
      [ Handlungsträger γ ermitteln Enddatum ]
      [ Handlungsträger γ festlegen Enddatum ]
    property abDatum: Object
      [ Handlungsträger γ ermitteln Startdatum ]
      [ Handlungsträger γ festlegen Startdatum ]
    property ort: String
      [ Handlungsträger γ ermitteln Ortskennung ]
      [ Handlungsträger γ festlegen Ortskennung ]
    property buchstabe: String
      [ Handlungsträger γ ermitteln Buchstabenfolge ]
      [ Handlungsträger γ festlegen Buchstabenfolge ]
    property zahl: String
      [ Handlungsträger γ ermitteln Ziffernfolge ]
      [ Handlungsträger γ festlegen Ziffernfolge ]

    getReservierung(out reservierung: Reservierung) {
      existing (ReservierungBeanC.Reservierung reservierung);
      return (reservierung);
    };
    setReservierung(in reservierung: Reservierung) {
      return;
    };
    getKfz(out kfz: Kfz) {
      existing (KfzBeanC.Kfz kfz);
      return (kfz);
    };
    setKfz(in kfz: Kfz) {
      return;
    };

    created()     =   initialized();
    initialized() =   setKfz?(kfz).vergeben()
                    + (getType?(type) + getOrt?(ort) + setOrt?(ort)
                       + setZahl?(zahl) + setBuchstabe?(buchstabe) + getId?(id)
                       + getBuchstabe?(buchstabe) + setType?(type) + setId?(id)
                       + getZahl?(zahl)).initialized()
                    + setReservierung?(reservierung).reserviert();
    vergeben()    =   (getOrt?(ort) + getBuchstabe?(buchstabe) + getZahl?(zahl)
                      + setAbDatum?(abDatum) + setBisDatum?(bisDatum)
                      + getBisDatum?(bisDatum) + getAbDatum?(abDatum)
                      + getId?(id) + getType?(type) + getKfz?(kfz)).vergeben()
                    + setKfz?(kfz).initialized();
    reserviert()  =   setKfz?(kfz).vergeben()
                    + setReservierung?(reservierung).initialized()
```

```
                    + (getReservierung?(reservierung) + getZahl?(zahl)
                      + getId?(id) + getOrt?(ort) + getType?(type)
                      + getBuchstabe?(buchstabe)).reserviert();
  };

  create(out kennzeichen: Kennzeichen)
    [ Handlungsträger γ anlegen Kennzeichen ] {
    new (KennzeichenBean.Kennzeichen kennzeichen);
    return (kennzeichen);
  };
  findByPrimaryKey(in key: String, out result: Set<KennzeichenBean.Kennzeichen>)
    [ Handlungsträger γ suchen Kennzeichen ] {
    existing(Set<KennzeichenBean.Kennzeichen> result);
    return(result);
  };
  remove(in key: String)
    [ Handlungsträger γ loeschen Kennzeichen ] {
    return;
  };

  created()     =   initialized();
  initialized() =   (findByPrimaryKey?(key, result)
                    + create?(kennzeichen) + remove?(key)).initialized();
};
```

Listing C.6: CDL-Beschreibung der Entity Bean `KennzeichenBean`

KfzBean Mit Hilfe der in Listing C.7 beschriebenen Entity Bean `KfzBean` werden u. a. Angaben zu Halter, Versicherung, Fahrzeug und Kennzeichen eines Kraftfahrzeugs verwaltet.

```
Component KfzBean {

  slot KennzeichenBeanC kennzeichenComp;
  slot VersicherungsdatenBeanC versicherungComp;
  slot AdresseBeanC adresseComp;
  slot HalterBeanC halterComp;
  slot FahrzeugdatenBeanC fahrzeugdatenComp;

  class Kfz {
    property id: String
      [ Handlungsträger γ ermitteln Primaerschluessel ]
      [ Handlungsträger γ festlegen Primaerschluessel ]
    property letzterUser: String
      [ Handlungsträger γ ermitteln Datenerfasser ]
      [ Handlungsträger γ festlegen Datenerfasser ]
    property letztesDatum: Object
      [ Handlungsträger γ ermitteln Bearbeitungsdatum ]
      [ Handlungsträger γ festlegen Bearbeitungsdatum ]
    property verfuegungsberechtigt: String
      [ Handlungsträger γ ermitteln Verfuegungsberechtigter ]
      [ Handlungsträger γ festlegen Verfuegungsberechtigter ]

    getStandort(out standort: Adresse) {
      existing (AdresseBeanC.Adresse adresse);
      return (adresse);
    };
    setStandort(in standort: Adresse) {
      return;
    };
```

```
  getHalter(out halter: Halter) {
    existing (HalterBeanC.Halter halter);
    return (halter);
  };
  setHalter(in halter: Halter) {
    return;
  };
  getFahrzeugDaten(out fahrzeugDaten: Fahrzeugdaten) {
    existing (FahrzeugdatenBeanC.Fahrzeugdaten fahrzeugdaten);
    return (fahrzeugdaten);
  };
  setFahrzeugDaten(in fahrzeugDaten: Fahrzeugdaten) {
    return;
  };
  getVersicherung(out versicherungsdaten: Versicherungsdaten) {
    existing (VersicherungsdatenBeanC.Versicherungsdaten versicherungsdaten);
    return (versicherungsdaten);
  };
  setVersicherung(in versicherung: Versicherungsdaten) {
    return;
  };
  getKennzeichen(out kennzeichen: Kennzeichen) {
    existing (KennzeichenBeanC.Kennzeichen kennzeichen);
    return (kennzeichen);
  };
  setKennzeichen(in kennzeichen: Kennzeichen) {
    return;
  };

  created()       =   initialized();
  initialized()   =   setId?(id).manipulierbar();
  manipulierbar() =   (setKennzeichen?(kennzeichen)
                      + setHalter?(halter)
                      + setFahrzeugDaten?(fahrzeugDaten)
                      + setVersicherung?(versicherung)
                      + setStandort?(standort)
                      + setLetztesDatum?(letztesDatum)
                      + setVerfuegungsberechtigt?(verfuegungsberechtigt)
                      + setLetzterUser?(letzterUser)
                      + getId?(id)
                      + getKennzeichen?(kennzeichen)
                      + getHalter?(halter)
                      + getFahrzeugDaten?(fahrzeugDaten)
                      + getVersicherung?(versicherungsdaten)
                      + getStandort?(standort)
                      + getLetztesDatum?(letztesDatum)
                      + getVerfuegungsberechtigt?(verfuegungsberechtigt)
                      + getLetzterUser?(letzterUser)).manipulierbar();
};

create(out kfz: Kfz)
  [ Handlungsträger γ anlegen Kfz ] {
  new (KfzBean.Kfz kfz);
  return (kfz);
};
findByPrimaryKey(in key: String, out result: Set<KfzBean.Kfz>)
  [ Handlungsträger γ suchen Kfz ] {
  existing(Set<KfzBean.Kfz> result);
  return(result);
};
remove(in key: String)
  [ Handlungsträger γ loeschen Kfz ] {
  return;
};
```

```
  created()     =   initialized();
  initialized() =   (findByPrimaryKey?(key, result)
                    + remove?(key) + create?(kfz)).initialized();
};
```

Listing C.7: CDL-Beschreibung der Entity Bean `KfzBean`

OFDDatenBean Die von der Oberfinanzdirektion für die Berechnung der fälligen Kraftfahrzeugsteuer erforderlichen Daten lassen sich mit der Entity Bean `OFDDatenBean` erfassen. Die CDL-Beschreibung dieser Komponente ist in Listing C.8 angegeben.

```
Component OFDDatenBean {

  slot KfzBeanC kfzComp;

  class OFDDaten {
    property id: String
      [ Handlungsträger γ ermitteln Primaerschluessel ]
      [ Handlungsträger γ festlegen Primaerschluessel ]
    property datum: Object
      [ Handlungsträger γ ermitteln Bearbeitungsdatum ]
      [ Handlungsträger γ festlegen Bearbeitungsdatum ]
    property zahlweise: int
      [ Handlungsträger γ ermitteln Zahlungsart ]
      [ Handlungsträger γ festlegen Zahlungsart ]
    property typ: String
      [ Handlungsträger γ ermitteln Typ ]
      [ Handlungsträger γ festlegen Typ ]
    property verguenstigung: String
      [ Handlungsträger γ ermitteln Verguenstigung ]
      [ Handlungsträger γ festlegen Verguenstigung ]
    property stufe: String
      [ Handlungsträger γ ermitteln Foerderstufe ]
      [ Handlungsträger γ festlegen Foerderstufe ]
    property anhaenger: String
      [ Handlungsträger γ ermitteln Anhaengerlast ]
      [ Handlungsträger γ festlegen Anhaengerlast ]
    property smogplakette: String
      [ Handlungsträger γ ermitteln Smogplakette ]
      [ Handlungsträger γ festlegen Smogplakette ]
    property angleichungsdatum: Object
      [ Handlungsträger γ ermitteln Zahlungsdatum ]
      [ Handlungsträger γ festlegen Zahlungsdatum ]
    property forderstufendatum: Object
      [ Handlungsträger γ ermitteln Foerderstufendatum ]
      [ Handlungsträger γ festlegen Foerderstufendatum ]
    property schadarmDatum: Object
      [ Handlungsträger γ ermitteln Schadstoffarmdatum ]
      [ Handlungsträger γ festlegen Schadstoffarmdatum ]

    getFahrzeug(out fahrzeug: Kfz) {
      existing (KfzBeanC.Kfz kfz);
      return (kfz);
    };
    setFahrzeug(in fahrzeug: Kfz) {
      return;
    };

    created()     =   initialized();
```

```
    initialized() =   setId?(id).id_gesetzt();
    id_gesetzt()  =   (getVerguenstigung?(verguenstigung)
                      + getAnhaenger?(anhaenger) + getStufe?(stufe)
                      + getAngleichungsdatum?(angleichungsdatum)
                      + getZahlweise?(zahlweise) + setTyp?(typ)
                      + setSmogplakette?(smogplakette) + setDatum?(datum)
                      + getSchadarmDatum?(schadarmDatum)
                      + setZahlweise?(zahlweise) + setAnhaenger?(anhaenger)
                      + setForderstufendatum?(forderstufendatum)
                      + getDatum?(datum) + setSchadarmDatum?(schadarmDatum)
                      + setAngleichungsdatum?(angleichungsdatum) + getId?(id)
                      + setFahrzeug?(fahrzeug) + getTyp?(typ)
                      + getForderstufendatum?(forderstufendatum)
                      + setVerguenstigung?(verguenstigung)
                      + getFahrzeug?(fahrzeug) + getSmogplakette?(smogplakette)
                      + setStufe?(stufe)).id_gesetzt();
  };

  create(out ofdDaten: OFDDaten)
    [ Handlungsträger γ anlegen OFDDaten ] {
    new (OFDDatenBean.OFDDaten ofdDaten);
    return (ofdDaten);
  };
  findByPrimaryKey(in key: String, out result: Set<OFDDatenBean.OFDDaten>)
    [ Handlungsträger γ suchen OFDDaten ] {
    existing(Set<OFDDatenBean.OFDDaten> result);
    return(result);
  };
  remove(in key: String)
    [ Handlungsträger γ loeschen OFDDaten ] {
    return;
  };


  created()     =   initialized();
  initialized() =   (remove?(key) + create?(ofdDaten)
                    + findByPrimaryKey?(key, result)).initialized();
};
```

Listing C.8: CDL-Beschreibung der Entity Bean `OFDDatenBean`

PersonBean Die Entity Bean `PersonBean`, die in Listing C.9 beschrieben ist, dient der Verwaltung personenbezogener Angaben, wie sie z. B. für Halter oder Versicherungsnehmer zu erfassen sind.

```
Component PersonBean {

  slot HalterBeanC halterComp;
  slot ReservierungBeanC reservierungComp;
  slot AdresseBeanC adresseComp;
  slot VersicherungsdatenBeanC versicherungComp;

  class Person {
    property id: String
      [ Handlungsträger γ ermitteln Primaerschluessel ]
      [ Handlungsträger γ festlegen Primaerschluessel ]
    property anrede: String
      [ Handlungsträger γ ermitteln Anrede ]
      [ Handlungsträger γ festlegen Anrede ]
```

```
property akaGrad: String
  [ Handlungsträger γ ermitteln AkademischerGrad ]
  [ Handlungsträger γ festlegen AkademischerGrad ]
property vorname: String
  [ Handlungsträger γ ermitteln Vorname ]
  [ Handlungsträger γ festlegen Vorname ]
property nachname: String
  [ Handlungsträger γ ermitteln Nachname ]
  [ Handlungsträger γ festlegen Nachname ]
property familienname: String
  [ Handlungsträger γ ermitteln Familienname ]
  [ Handlungsträger γ festlegen Familienname ]
property firmenname: String
  [ Handlungsträger γ ermitteln Firmenname ]
  [ Handlungsträger γ festlegen Firmenname ]
property geburtsname: String
  [ Handlungsträger γ ermitteln Geburtsname ]
  [ Handlungsträger γ festlegen Geburtsname ]
property geburtsdatum: String
  [ Handlungsträger γ ermitteln Geburtsdatum ]
  [ Handlungsträger γ festlegen Geburtsdatum ]
property geburtsort: String
  [ Handlungsträger γ ermitteln Ort ]
  [ Handlungsträger γ festlegen Ort ]
property vorwahl: String
  [ Handlungsträger γ ermitteln Vorwahl ]
  [ Handlungsträger γ festlegen Vorwahl ]
property telNr: String
  [ Handlungsträger γ ermitteln Telefonnummer ]
  [ Handlungsträger γ festlegen Telefonnummer ]
property blz: String
  [ Handlungsträger γ ermitteln Bankleitzahl ]
  [ Handlungsträger γ festlegen Bankleitzahl ]
property kontoNr: String
  [ Handlungsträger γ ermitteln Kontonummer ]
  [ Handlungsträger γ festlegen Kontonummer ]
property typ: int
  [ Handlungsträger γ ermitteln Typ ]
  [ Handlungsträger γ festlegen Typ ]

getAdresse(out adresse: Adresse) {
  existing (AdresseBeanC.Adresse adresse);
  return (adresse);
};
setAdresse(in adresse: Adresse) {
  return;
};
getHalter(out halter: Halter) {
  existing (HalterBeanC.Halter halter);
  return (halter);
};
setHalter(in halter: Halter) {
  return;
};
getReservierung(out reservierung: Reservierung) {
  existing (ReservierungBeanC.Reservierung reservierung);
  return (reservierung);
};
setReservierung(in reservierung: Reservierung) {
  return;
};
getVersicherungen
    (out versicherungen: Set<VersicherungsDatenBeanC.VersicherungsDaten>) {
  existing(Set<VersicherungsDatenBeanC.VersicherungsDaten> versicherungen);
```

```
      return (versicherungen);
    };
    setVersicherungen
        (in versicherungen: Set<VersicherungsDatenBeanC.VersicherungsDaten>) {
      return;
    };

    created()     =   initialized();
    initialized() =   setId?(id).id_gesetzt();
    id_gesetzt()  =  (setAdresse?(adresse) + getTelNr?(telNr)
                     + getHalter?(halter) + setNachname?(nachname)
                     + getReservierung?(reservierung) + getAnrede?(anrede)
                     + setKontoNr?(kontoNr) + setFirmenname?(firmenname)
                     + setFamilienname?(familienname) + getVorwahl?(vorwahl)
                     + getGeburtsname?(geburtsname) + setTelNr?(telNr)
                     + setAkaGrad?(akaGrad) + getVorname?(vorname)
                     + getGeburtsdatum?(geburtsdatum) + setVorname?(vorname)
                     + setVersicherungen?(versicherungen) + setHalter?(halter)
                     + setVorwahl?(vorwahl) + getId?(id) + getKontoNr?(kontoNr)
                     + setReservierung?(reservierung)
                     + setGeburtsname?(geburtsname) + getFirmenname?(firmenname)
                     + getNachname?(nachname) + setTyp?(typ) + getTyp?(typ)
                     + getGeburtsort?(geburtsort) + getAkaGrad?(akaGrad)
                     + getBlz?(blz) + setGeburtsort?(geburtsort)
                     + getVersicherungen?(versicherungen) + setBlz?(blz)
                     + setAnrede?(anrede) + getFamilienname?(familienname)
                     + getAdresse?(adresse) + setGeburtsdatum?(geburtsdatum))
                     .id_gesetzt();
  };

  create(out person: Person)
    [ Handlungsträger γ anlegen Person ] {
    new (PersonBean.Person person);
    return (person);
  };
  findByPrimaryKey(in key: String, out result: Set<PersonBean.Person>)
    [ Handlungsträger γ suchen Person ] {
    existing(Set<PersonBean.Person> result);
    return(result);
  };
  remove(in key: String)
    [ Handlungsträger γ loeschen Person ] {
    return;
  };

  created()     =   initialized();
  initialized() =   (findByPrimaryKey?(key, result)
                    + create?(person) + remove?(key)).initialized();
};
```

Listing C.9: CDL-Beschreibung der Entity Bean `PersonBean`

ReservierungBean Reservierungen von Kennzeichen lassen sich mit Hilfe der Entity Bean `ReservierungBean` persistent speichern. Listing C.10 zeigt die CDL-Beschreibung dieser Komponente.

```
Component ReservierungBean {

  slot KennzeichenBeanC kennzeichenComp;
  slot PersonBeanC personComp;
```

```
  class Reservierung {
    property id: String
      [ Handlungsträger γ ermitteln Primaerschluessel ]
      [ Handlungsträger γ festlegen Primaerschluessel ]
    property reservierungAm: Object
      [ Handlungsträger γ ermitteln Startdatum ]
      [ Handlungsträger γ festlegen Startdatum ]
    property reservierungsEnde: Object
      [ Handlungsträger γ ermitteln Enddatum ]
      [ Handlungsträger γ festlegen Enddatum ]

    getPerson(out person: Person) {
      existing (PersonBeanC.Person person);
      return (person);
    };
    setPerson(in person: Person) {
      return;
    };
    getKennzeichen(out kennzeichen: Kennzeichen) {
      existing (KennzeichenBeanC.Kennzeichen kennzeichen);
      return (kennzeichen);
    };
    setKennzeichen(in kennzeichen: Kennzeichen) {
      return;
    };

    created()     =   initialized();
    initialized() =   setId?(id).id_gesetzt();
    id_gesetzt()  =   (getReservierungAm?(reservierungAm)
                      + getKennzeichen?(kennzeichen)
                      + getReservierungsEnde?(reservierungsEnde)
                      + setReservierungAm?(reservierungAm)
                      + setReservierungsEnde?(reservierungsEnde)
                      + getPerson?(person) + setPerson?(person)
                      + setKennzeichen?(kennzeichen) + getId?(id))
                      .id_gesetzt();
  };

  create(out reservierung: Reservierung)
    [ Handlungsträger γ anlegen Reservierung ] {
    new (ReservierungBean.Reservierung reservierung);
    return (reservierung);
  };
  findByPrimaryKey(in key: String,
                   out result: Set<ReservierungBean.Reservierung>)
    [ Handlungsträger γ suchen Reservierung ] {
    existing(Set<ReservierungBean.Reservierung> result);
    return(result);
  };
  remove(in key: String)
    [ Handlungsträger γ loeschen Reservierung ] {
    return;
  };

  created()     =   initialized();
  initialized() =   (findByPrimaryKey?(key, result)
                    + create?(reservierung) + remove?(key)).initialized();
};
```

Listing C.10: CDL-Beschreibung der Entity Bean `ReservierungBean`

VersicherungsdatenBean Mit Hilfe der Entity Bean `VersicherungsdatenBean` lassen sich alle Daten ablegen, die im Rahmen der Kfz-Neuzulassung bzgl. der Versicherung eines Fahrzeugs zu erfassen sind.

```
Component VersicherungsdatenBean {

  slot PersonBeanC personComp;
  slot KfzBeanC kfzComp;

  class Versicherungsdaten {
    property versicherungsNr: String
      [ Handlungsträger γ ermitteln Versicherungsnummer ]
      [ Handlungsträger γ festlegen Versicherungsnummer ]
    property agentur: String
      [ Handlungsträger γ ermitteln Versicherung ]
      [ Handlungsträger γ festlegen Versicherung ]
    property doppelkarte: String
      [ Handlungsträger γ ermitteln Doppelkarte ]
      [ Handlungsträger γ festlegen Doppelkarte ]
    property versicherungsscheinNr: String
      [ Handlungsträger γ ermitteln Versicherungsscheinnummer ]
      [ Handlungsträger γ festlegen Versicherungsscheinnummer ]
    property deckungssumme: String
      [ Handlungsträger γ ermitteln Deckungssumme ]
      [ Handlungsträger γ festlegen Deckungssumme ]
    property interneVermerke: String
      [ Handlungsträger γ ermitteln Vermerk ]
      [ Handlungsträger γ festlegen Vermerk ]
    property ungestempeltesAKZ: boolean
      [ Handlungsträger γ ermitteln FahrtenUngueltigesAkz ]
      [ Handlungsträger γ festlegen FahrtenUngueltigesAkz ]
    property versicherungsbeginn: Object
      [ Handlungsträger γ ermitteln Startdatum ]
      [ Handlungsträger γ festlegen Startdatum ]
    property versicherungsEnde: Object
      [ Handlungsträger γ ermitteln Enddatum ]
      [ Handlungsträger γ festlegen Enddatum ]
    property nation: String
      [ Handlungsträger γ ermitteln Nation ]
      [ Handlungsträger γ festlegen Nation ]

    getKfz(out kfz: Object) {
      return (*);
    };
    setKfz(in kfz: Object) {
      return;
    };
    getPerson(out person: Person) {
      existing (PersonBeanC.Person person);
      return (person);
    };
    setPerson(in person: Person) {
      return;
    };

    created()     =   initialized();
    initialized() =   (setVersicherungsscheinNr?(versicherungsscheinNr)
                      + getUngestempeltesAKZ?(ungestempeltesAKZ)
                      + getInterneVermerke?(interneVermerke)
                      + getAgentur?(agentur) + setKfz?(kfz)
                      + setDeckungssumme?(deckungssumme)
                      + getVersicherungsbeginn?(versicherungsbeginn)
                      + setVersicherungsEnde?(versicherungsEnde)
```

```
                        + getVersicherungsscheinNr?(versicherungsscheinNr)
                        + setNation?(nation) + setDoppelkarte?(doppelkarte)
                        + setUngestempeltesAKZ?(ungestempeltesAKZ)
                        + setVersicherungsNr?(versicherungsNr) + getPerson?(person)
                        + getVersicherungsEnde?(versicherungsEnde)
                        + getVersicherungsNr?(versicherungsNr)
                        + setInterneVermerke?(interneVermerke) + getNation?(nation)
                        + setAgentur?(agentur) + getKfz?(kfz)
                        + getDeckungssumme?(deckungssumme)
                        + getDoppelkarte?(doppelkarte) + setPerson?(person)
                        + setVersicherungsbeginn?(versicherungsbeginn))
                        .initialized();
  };

  create(out versicherungsdaten: Versicherungsdaten)
    [ Handlungsträger γ anlegen Versicherungsdaten ] {
    new (VersicherungsdatenBean.Versicherungsdaten versicherungsdaten);
    return (versicherungsdaten);
  };
  findByPrimaryKey(in key: String,
                   out result: Set<VersicherungsdatenBean.Versicherungsdaten>)
    [ Handlungsträger γ suchen Versicherungsdaten ] {
    existing(Set<VersicherungsdatenBean.Versicherungsdaten>
             result);
    return(result);
  };
  remove(in key: String)
    [ Handlungsträger γ loeschen Versicherungsdaten ] {
    return;
  };

  created()     =   initialized();
  initialized() =   (create?(versicherungsdaten) + findByPrimaryKey?(key, result)
                    + remove?(key)).initialized();
};
```

Listing C.11: CDL-Beschreibung der Entity Bean `VersicherungsdatenBean`

Glossar

Dieses Glossar definiert wichtige in dieser Arbeit verwendete Begriffe. Diese entstammen vornehmlich den Bereichen „Komponentensoftware“, „Geschäftsprozessmodellierung“ und „formale/objektorientierte Softwareentwicklung“.

Ablauforganisation Teil der Organisation eines Unternehmens, der das zeitlich-logische (dynamische) Verhalten von Vorgängen behandelt, die der Aufgabenerfüllung des Unternehmens dienen (nach [Sch98b]).

Aktion Atomares Element des Verhaltens eines Systems. Eine ∼ hat keine Dauer und kann je nach Typ für einen Beobachter des Systems sichtbar oder unsichtbar (τ-Aktion) sein.

(Betriebliche) Aktivität Schritt eines →Geschäftsprozesses, mit dem ein bestimmtes unternehmensrelevantes Ziel verfolgt wird. Eine ∼ beinhaltet den auszuführenden Vorgang, die dafür benutzten, manipulierten oder erzeugten Ressourcen sowie die für die Ausführung verantwortliche Organisationseinheit.

Application Server Softwareprodukt, das eine →Komponententechnologie implementiert. Synonym auch Bezeichnung einer konkreten Installation eines solchen Softwareprodukts auf einem zentral verfügbaren Rechner.

Architekturbeschreibungssprache Sprache zur abstrakten Beschreibung der Architekturen komplexer Softwaresysteme durch die Darstellung ihres Aufbaus aus einzelnen, miteinander interagierenden →Komponenten (vgl. auch →Komponentenbeschreibungssprache). Vielfach gestatten ∼n auch die abstrakte Spezifikation des Verhaltens der Komponenten sowie des gesamten Softwaresystems.

Aufbauorganisation Teil der Organisation eines Unternehmens, der Leitungs-, Leistungs-, Informations- und Kommunikationsbeziehungen zwischen Unternehmensteilen durch (statische) Regelungen wie Hierarchien oder Unternehmenstopologien behandelt (nach [Sch98b]).

Begriff Wort, dessen Bedeutung durch exemplarische Einführung, Prädikatorenregeln oder explizite Definition inhaltlich geklärt und definiert worden ist.

Customizing Prozess der Anpassung (Parametrisierung/Konfiguration) eines standardisierten betrieblichen Anwendungssoftwareprodukts (→Standardsoftware) an die konkreten Anforderungen eines Unternehmens, das Kunde („Customer“) des Herstellers dieses Softwareprodukts ist (nach [AR00]).

Design by Contract Softwareentwicklungsmethode, bei der Softwareelemente als Anbieter und Nutzer von Diensten auf der Grundlage von (formalen) Verträgen miteinander kooperieren [Mey92]. Zentraler Bestandteil dieser Verträge ist die Spezifikation von Vor- und Nachbedingungen der Nutzung bereitgestellter Dienste.

Extension Menge der Objekte, die unter einem →Begriff subsumiert werden (Umfang eines Begriffs, vgl. auch →Intension). Die ∼ eines Begriffs bildet die Menge aller Objekte, denen der Begriff im Sinne der →Prädikation zugeschrieben werden kann.

Fachkomponente →Komponente, die „vertikale“ Dienste mit einer fachlichen, z. B. betriebswirtschaftlichen Anwendungslogik anbietet (vgl. auch →GUI-Komponente und →Systemkomponente). Abhängig vom primären Einsatzgebiet einer ∼ lassen sich →Geschäftsobjektkomponenten und →Prozesskomponenten unterscheiden.

Frame Begriff aus der künstlichen Intelligenz zur Bezeichnung einer Zusammenstellung von Wissen, das für ein Objekt, Konzept oder in einer bestimmten Situation von Bedeutung ist. Ein ∼ besteht gewöhnlich aus einem Namen und einer Liste von Attribut/Wert-Paaren. Die Attribute werden auch Slots und ihre Werte Fillers genannt (nach [Tan90]).

Framework Wiederverwendbare, „halbfertige“ Anwendung, die Struktur und Verhalten eines Entwurfs vorgibt und an so genannten Hot Spots erweitert werden muss, um eine fertige, kundenindividuell angepasste Anwendung zu entwickeln. Hinsichtlich der für die Erweiterung eines Frameworks eingesetzten Technik lassen sich →Klassenframeworks und →Komponentenframeworks unterscheiden.

Geschäftsobjekt Allgemein eine Instanz eines (abstrakten) betrieblichen Konzepts in der Informationsverarbeitung (z. B. ein spezifischer Auftrag). Speziell eine (persistente) Instanz eines durch eine →Geschäftsobjektkomponente repräsentierten Konzepts. Ein ∼ kann softwaretechnisch als →(aktives) Objekt realisiert sein.

Geschäftsobjektkomponente →Komponente, die ein zentrales Konzept des betrachteten Anwendungsbereichs wie z. B. „Auftrag“ oder „Kunde“ repräsentiert und Dienste für den Zugriff auf →Geschäftsobjekte dieses Konzepts anbietet (vgl. auch →Prozesskomponente).

Geschäftsprozess Zeitlich und sachlogisch zusammenhängende, möglicherweise organisationseinheitenübergreifende Menge betrieblicher →Aktivitäten, die ein bestimmtes unternehmensrelevantes Ziel verfolgen und zur Bearbeitung auf Unternehmensressourcen zurückgreifen (nach [Rum99]).

Geschäftsprozessinstanz Konkrete Ausführung eines →Geschäftsprozesses.

Geschäftsprozessmodell Repräsentation eines Geschäftsprozesses auf der Grundlage einer →Geschäftsprozessmodellierungssprache.

Geschäftsprozessmodellierung Vorgang der Beschreibung von →Geschäftsprozessen durch die Erstellung von →Geschäftsprozessmodellen.

Geschäftsprozessmodellierungssprache Sprache zur Beschreibung von →Geschäftsprozessen. ∼n werden verbreitet als Diagrammsprachen angelegt, um eine höhere Verständlichkeit für Fachexperten zu erzielen.

GUI-Komponente Visuelle →Komponente, die der Konstruktion graphischer Benutzungsoberflächen dient und auf Clientseite eingesetzt wird (vgl. auch →Fach- und →Systemkomponente).

Individualsoftware Anwendungssoftware, die individuell für ein Unternehmen unter Berücksichtigung seiner spezifischen Anforderungen entwickelt wird (vgl. auch →Standardsoftware).

Intension Menge der Merkmale, die einem →Begriff zukommen (Bedeutung eines Begriffs). Anhand der ∼ lässt sich überprüfen, ob ein Objekt Teil der →Extension eines Begriffs ist.

Klasse Statische Beschreibung, die den Zustandsraum und das Verhalten einer Menge von →Objekten (ihren Instanzen) definiert. Der Zustandsraum wird dabei durch Attribute spezifiziert, während das Verhalten durch Methoden gegeben ist. Eine ∼ kann Attribute und Methoden von einer anderen ∼ erben (→Vererbung).

Klassenframework (auch objektorientiertes Framework) →Framework, das mittels Vererbung angepasst wird, indem kundenspezifische →Klassen von den (abstrakten) Klassen des ∼s abgeleitet werden. ∼s werden aufgrund dieses Erweiterungsmechanismus auch White-Box-Frameworks genannt.

Komponente Abstraktions- und Kompositionseinheit, die eine in sich geschlossene, vermarktbare Softwarelösung für einen abgegrenzten Problembereich anbietet. ∼n werden als ausführbare Einheiten zur Verfügung gestellt, die aufgrund expliziter Spezifikationen von Schnittstellen und Kontextabhängigkeiten im Sinne einer Black-Box-→Wiederverwendung durch Dritte genutzt werden können.

Komponentenbasierte Softwareentwicklung Ansatz zur Entwicklung bzw. Anpassung von →Komponentensoftware durch die Komposition von →Komponenten auf der Grundlage von →Komponentenframeworks. Dabei können neben neu entwickelten Komponenten auch vorgefertigte Komponenten zum Einsatz kommen, die möglicherweise von unterschiedlichen Herstellern stammen.

Komponentenbeschreibung Abstrakte Beschreibung einer →Komponente, die deren Funktionalität und Anforderungen sowohl auf syntaktischer (technischer) als auch auf semantischer (fachlicher) Ebene spezifiziert. Neben der rein syntaktischen Spezifikation von Schnittstellen dokumentiert eine ∼ im engeren, in dieser Arbeit verwendeten Sinne insbesondere auch das abstrakte Verhalten einer Komponente durch die Angabe von Lebenszyklen (formale →Semantik) und die fachliche Bedeutung der von ihr angebotenen Dienste (fachliche →Semantik).

Komponentenbeschreibungssprache Sprache zur Formulierung von →Komponentenbeschreibungen (vgl. auch →Architekturbeschreibungssprache).

(Komponenten-)Broker Rolle eines Akteurs im Makroprozess geschäftsprozessorientierter Komponentensuche, der →Komponentenverwender bei der Suche nach geeigneten →Komponenten durch semantische Suchdienste unterstützt.

Komponentenframework →Framework, das durch Komposition mit kundenspezifischen →Komponenten erweitert werden kann. Die Erweiterung eines ∼s mit Komponenten wird kontrolliert, indem so genannte Erweiterungspunkte mit →Kontrakten typisiert werden. ∼s werden aufgrund des verwendeten Erweiterungsmechanismus auch als Black-Box-Frameworks bezeichnet.

(Komponenten-)Hersteller Rolle eines Akteurs im Makroprozess geschäftsprozessorientierter Komponentensuche, der →Komponenten entwickelt und auf einem →Komponentenmarkt anbietet.

Komponenteninstanz →Komponenten als solche können in der Regel nicht instanziiert werden. Sie realisieren jedoch im Allgemeinen eine Menge instanziierbarer Abstraktionen (z. B. →Klassen von →Geschäftsobjekten). Ein Netz von Instanzen solcher Abstraktionen, das ein konzeptionelles Ganzes formt (z. B. →Objekte, die einen Auftrag repräsentieren), wird gelegentlich als ∼ bezeichnet (nach [Szy02]).

Komponentenmarkt Elektronischer Markplatz, der als Integrationsplattform für angebotene →Komponenten, Marktteilnehmer sowie verfügbare Dienstleistungen dient.

Komponentenmodell Rahmen für die Entwicklung und den Einsatz von →Komponenten, der strukturelle Anforderungen hinsichtlich der Komposition sowie verhaltensorientierte Anforderungen hinsichtlich der Kollaboration von Komponenten definiert. Ein ∼ stellt darüber hinaus Anforderungen an eine Infrastruktur für den Einsatz von Komponenten, zu denen verbreitet die Bereitstellung von Mechanismen für Verteilung, Persistenz, Nachrichtenaustausch, Sicherheit und Versionierung gehören.

Komponentensoftware Software, die durch die kundenindividuelle Komposition und Anpassung von (standardisierten) Komponenten entwickelt wird, um die Vorteile von →Individual- und →Standardsoftware miteinander zu kombinieren.

Komponententechnologie Konkrete Spezifikation einer Infrastruktur für den Einsatz von →Komponenten eines bestimmten →Komponentenmodells. Populäre Beispiele für ∼n sind COM+, .NET und Enterprise Java Beans (EJB).

(Komponenten-)Verwender Rolle eines Akteurs im Makroprozess geschäftsprozessorientierter Komponentensuche, der →Komponenten auf einem →Komponentenmarkt erwirbt und im operativen Einsatz oder für die Entwicklung komplexer Komponenten nutzt.

Kontrakt Vertragsähnliche Vereinbarung zwischen einem Anbieter eines Dienstes und dessen Nutzer, die neben funktionalen Aspekten auch nicht-funktionale Aspekte des Dienstes beinhalten kann. Zu den funktionalen Aspekten eines Kontrakts gehören z. B. Syntax und Semantik der erwarteten Schnittstellen. Nicht-funktionale Aspekte beziehen sich z. B. auf die Zusicherung bestimmter Qualitätseigenschaften.

Kontravarianz Ersetzung des Typs eines Methodenparameters durch einen Subtyp (bzw. Supertyp) dieses Typs beim Übergang von der Schnittstelle eines Typs zu der Schnittstelle eines Supertyps (bzw. Subtyps) diesen Typs (vgl. auch →Kovarianz).

Kovarianz Ersetzung des Typs eines Methodenparameters durch einen Subtyp (bzw. Supertyp) beim Übergang von der Schnittstelle eines Typs zu der Schnittstelle eines Subtyps (bzw. Supertyps) diesen Typs (vgl. auch →Kontravarianz).

Lebendigkeit Eigenschaft von Software, die die (sinnvolle) Fortsetzung ihrer Ausführung in jedem erreichbaren Zustand gewährleistet (vgl. auch →Sicherheit).

Marktplatzbetreiber Rolle eines Akteurs im Makroprozess geschäftsprozessorientierter Komponentensuche, der die Infrastruktur zur Verwaltung von →Komponenten bzw. →Komponentenbeschreibungen zur Verfügung stellt (siehe auch →Repository).

Metadaten Jede Form von Informationen über Daten. Dazu gehören im Umfeld der →Komponentensoftware Beschreibungen von →Komponenten (→Komponentenbeschreibungen).

Nichtdeterminismus Eigenschaft eines Algorithmus, von einem festen Anfangszustand aus durch willkürliche Auswahl einer von mehreren alternativen Fortsetzungsmöglichkeiten verschiedene Berechnungen durchführen zu können, die auch zu verschiedenen Endzuständen führen können (nach [AO94]).

Normierungsgremium Rolle eines Akteurs im Makroprozess geschäftsprozessorientierter Komponentensuche, der fachliche und technische Standards definiert, um die Interoperabilität von Komponenten zu fördern und die Suche nach Komponenten zu unterstützen.

Normsprache Sprache mit rationaler Grammatik und kontrolliertem Wortschatz, die der konstruktiven Wissenschaftstheorie folgend durch methodische Rekonstruktion der in einem Anwendungsbereich gesprochenen natürlichen (Fach-)Sprache entwickelt wird.

(Aktives) Objekt In der objektorientierten Softwareentwicklung eine Entität mit eindeutiger Identität, die als Instanz einer →Klasse ihren Zustand in Attributen verwaltet und ihr Verhalten über Methoden anbietet. Ein ∼ heißt aktiv, wenn es als autonome Einheit einen eigenen Kontrollfluss definiert und mit anderen ∼en durch den Austausch von Nachrichten kommuniziert. Da die Bereitschaft zur Kommunikation bei aktiven ∼en durch spezielle Interaktionsprotokolle gesteuert wird, sind im Allgemeinen nicht alle Dienste eines aktiven ∼s zu jedem Zeitpunkt verfügbar (non-uniform service availability).

Ontologie Explizite Spezifikation einer Konzeptualisierung eines betrachteten Weltausschnitts. Dabei bezeichnet der Begriff der Konzeptualisierung eine intensionale Struktur, die die relevanten Begriffe des Weltausschnitts sowie deren Beziehungen untereinander wiedergibt. Eine Unternehmens∼ ist eine ∼ zur Beschreibung der relevanten Konzepte von Unternehmen.

Parametrisierung Konfiguration eines →Komponentenframeworks durch Zuordnung von →Komponenten zu dessen Erweiterungspunkten.

Prädikation Unterscheidungshandlung, die nicht-sprachliches und sprachliches Handeln verbindet, indem einem Gegenstand(-styp) eine Eigenschaft zu- oder abgesprochen wird.

Prozessalgebra Formalisierte mathematische Sprache, die zum einen die Formulierung von →Prozesstermen durch die Bereitstellung von Konstanten sowie Operatoren und zum anderen deren Vergleich durch die Definition von Axiomen unterstützt.

Prozesskomponente →Komponente, die fachlich relevante, oftmals komplexere Dienste für die Abwicklung von Geschäftsprozessen wie z. B. die Auftragsbearbeitung bereitstellt (vgl. auch →Geschäftsobjektkomponente).

Prozessterm Linearer Ausdruck, der auf Grundlage der durch eine →Prozessalgebra definierten Konstanten und Operatoren gebildet wird.

Repository Datenbanksystem zur Verwaltung von →Metadaten (auch Metainformationssystem). Ein Komponenten-∼ verwaltet entsprechend Metadaten von →Komponenten.

Semantik Inhaltliche Bedeutung eines syntaktischen (sprachlichen) Konstrukts. Wir unterscheiden zwischen formaler ∼, die einem sprachlichen Konstrukt ein abstraktes (Maschinen-)Modell zuweist, und fachlicher ∼, die einem solchen abstrakten Modell durch Beschreibung der fachlichen Bezüge eine konkrete Interpretation zuordnet.

Sicherheit Eigenschaft von Software, die gewährleistet, dass in keinem erreichbaren Zustand unzulässige Fortsetzungen ihrer Ausführung möglich sind (vgl. auch →Lebendigkeit).

Softwarereferenzmodell Abstrakte Beschreibung eines (betrieblichen) Anwendungssoftwareprodukts, das die durch das Produkt unterstützten Funktionen, Abläufe und Strukturen aus einer sachlogischen und fachlichen Perspektive beschreibt. ∼e stellen vielfach eine Grundlage für das →Customizing dar.

Standardsoftware Anwendungssoftware, die aufgrund umfangreicher Anpassungsmöglichkeiten (→Customizing) sowie der Berücksichtigung von gesetzlichen, normierenden und De-Facto-Standards die Anforderungen möglichst vieler Unternehmen abdecken soll (vgl. auch →Individualsoftware).

Substituierbarkeit Eigenschaft eines Objekts, ein anderes Objekt, das im Allgemeinen von einem anderen →Typ ist, ersetzen zu können, ohne dass Clients des ersetzten Objekts diese Ersetzung feststellen (nach [Szy02]).

(Behavioural) Subtyping Ermittlung bzw. Herstellung von Subtyp-Beziehungen zwischen →Typen. Das ∼ ist eine formale Technik zur Unterstützung der →Substituierbarkeit von Objekten anhand ihrer Typen. Wird neben den Signaturen auch das Verhalten von Typen betrachtet, spricht man vom Behavioural ∼.

Systemkomponente →Komponente, die „horizontale“, also vom betrachteten Anwendungsbereich unabhängige Dienste bereitstellt (vgl. auch →GUI-Komponente und →Fachkomponente).

Terminologie System von →Begriffen, das eine mögliche sprachliche Repräsentation einer →Ontologie darstellt.

(Beschriftetes) Transitionssystem Gerichteter Graph, dessen Knoten Zustände repräsentieren und dessen Kanten mit dem Namen der →Aktion beschriftet sind, die bei dem jeweiligen Zustandsübergang ausgeführt wird.

Typ Menge von →Objekten, denen gewisse gemeinsame, extern beobachtbare Eigenschaften innewohnen. Ein Subtyp eines ∼s weist dessen Eigenschaften auf, die er durch weitere, nicht konkurrierende Eigenschaften ergänzen kann. Letztgenannter ∼ wird dann auch als Supertyp bezeichnet.

Vererbung Weitergabe von Schnittstellen oder Implementierung von einer →Klasse (der Oberklasse) an eine andere (die Unterklasse). ∼ unter Berücksichtigung gewisser Verfeinerungsregeln wie z. B. →Kovarianz der Typen von Eingabeparametern und →Kontravarianz der Typen von Ausgabeparametern und Rückgabewerten führt zum Begriff des →(Behavioural) Subtyping.

Deadlock Konflikt beim wechselseitigen Zugriff auf gemeinsam genutzte Ressourcen (z. B. Variablen, Betriebsmittel), bei dem keiner der Konfliktpartner seine Verarbeitung fortsetzen kann, weil er auf die Freigabe einer Ressource durch einen anderen Konfliktpartner wartet.

Wiederverwendung Wiederholte Anwendung verschiedenartigen Wissens über ein System auf ein anderes, ähnliches System mit dem Ziel, den Entwicklungs- und Wartungsaufwand des anderen Systems zu reduzieren (nach [BP89]). Dabei beinhaltet das wiederverwendete Wissen u. a. Domänenwissen, Entwürfe, Architekturen, Anforderungen, Quellcode und Dokumentation. Während die ∼ durch Vererbung (*White-Box-Wiederverwendung*) Zugang zum Quellcode erfordert, stützt sich die ∼ durch Komposition (*Black-Box-Wiederverwendung*) nur auf Schnittstellen ab.

Abkürzungsverzeichnis

AECMA	European Association of Aerospace Industries
ACE	Attempto Controlled English
ADL	Architecture Description Language
API	Application Programming Interface
ARIS	Architektur integrierter Informationssysteme
BALES	Binding Business Applications to LEgacy Systems
BPEL4WS	Business Process Execution Language for Web Services
CCM	CORBA Component Model
CDL	Component Description Language, auch Component Definition Language
CHARS	Corporate History Analyzer
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CTE	Caterpillar Technical English
CWM	Common Warehouse Model
DBMS	Database Management System
DCE	Distributed Computing Environment
DTD	Document Type Definition
EBNF	Extended Backus Naur Form
eEPK	Erweiterte ereignisgesteuerte Prozesskette
ECOOP	European Conference for Object-Oriented Programming
EJB	Enterprise JavaBeans
EPK	Ereignisgesteuerte Prozesskette
ERM	Entity-Relationship-Modell
ERP	Enterprise Resource Planning
FORWIN	Bayerischer Forschungsverbund Wirtschaftsinformatik

GI	Gesellschaft für Informatik
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDL	Interface Definition Language
ISO	International Organization for Standardization
IT	Informationstechnologie
J2EE	Java 2 Enterprise Edition
JML	Java Modeling Language
JSP	JavaServer Page(s)
KAON	Karlsruhe Ontology and Semantic Web Tool Suite
LTS	Labelled Transition System (engl. für „beschriftetes Transitionssystem“)
MOF	Meta Object Facility
NATO	North Atlantic Treaty Organisation
OAG	Open Applications Group
OASIS	Organization for the Advancement of Structured Information Standards
OCL	Object Constraint Language
oEPK	Objektorientierte ereignisgesteuerte Prozesskette
OFFIS	Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme
OIM	Open Information Model
OMG	Object Management Group
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering
OSF	Open Software Foundation
PSL	Process Specification Language
pUML	Precise UML
REBOOT	REuse Based on Object-Oriented Techniques
RDF	Resource Description Framework
SCDL	Simple Component Description Language
SOS	Strukturierte Operationelle Semantik
TOVE	TOronto Virtual Enterprise
UDDI	Universal Description, Discovery, and Integration
UML	Unified Modeling Language
VDMA	Verband deutscher Maschinen- und Anlagenbau
WSCI	Web Service Choreography Interface
WSDL	Web Service Description Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Symbolverzeichnis

$\mathbb{N}$	Menge der natürlichen Zahlen ohne 0
a, b, $c, \ldots$	(sichtbare) Aktionen
τ	unsichtbare Aktion
$\surd$	Terminierungsaktion
Σ	Menge aller sichtbaren Aktionen
$\Sigma_\tau = \Sigma \cup \{\tau\}$	Menge aller Aktionen
$\sigma \in \Sigma_\tau^*$	Wort
$\downarrow$	Restriktion
Θ	Menge aller einfachen Zustände
$A \subseteq \Sigma$	Menge von sichtbaren Aktionen
$Q \subseteq \Theta$	Menge von Zuständen
$q_0 \in Q$	Startzustand
$\mathcal{T}$	Einfaches (beschriftetes) Transitionssystem
$\alpha(\mathcal{T})$	Alphabet des einfachen beschrifteten Transitionssystems $\mathcal{T}$
LTS	Menge aller einfachen beschrifteten Transitionssysteme
$\mathcal{T}_\mathcal{X}$	Komplexes (beschriftetes) Transitionssystem
$\Pi_{\mathcal{T}_\mathcal{X}}$	Menge komplexer Zustände eines komplexen Transitionssystems $\mathcal{T}_\mathcal{X}$
$\rightarrow$	Transitionsrelation
$\Rightarrow$	Transitionsrelation mit Berücksichtigung von $\{\tau\}^*$-Präfixen
$\dashrightarrow$	Verweisrelation zwischen einfachen Transitionssystemen
$\backslash_h$	Hiding-Operator
$\sim_{e\tau}$	Embedded Weak Simulation
$\sqsubseteq_{match}$	Subtyping-Relation *Match*

P, Q	Prozessterme
ϵ	leerer Prozessterm
Ω	terminierter Prozessterm
$LProc$	Menge linearer Prozessterme
$\mathcal{S}(\cdot,\cdot,\cdots)$	Sequentielle Komposition der Prozessalgebra linearer Prozessmodelle
$\mathcal{A}(\cdot,\cdot,\cdots)$	Alternative Komposition der Prozessalgebra linearer Prozessmodelle
$\mathcal{P}(\cdot,\cdot,\cdots)$	Parallele Komposition der Prozessalgebra linearer Prozessmodelle
$\mathcal{L}(\cdot)$	Iteration der Prozessalgebra linearer Prozessmodelle
$\mathcal{SOS}_{\mathcal{LP}}(\cdot)$	Strukturierte operationelle Semantik linearer Prozessmodelle
N	Nominator
Q	Dinghauptprädikator
P	Geschehnisprädikator
ϵ	Seinskopula
ν	Teilungskopula
π	Tatkopula
γ	Fähigkeitskopula
!	Appellator
T_K	Menge von Konzepten
T_A	Menge von Aktivitäten
T	Terminologie
$\leq_K$	Spezialisierungsrelation über Konzepten
$\leq_A$	Spezialisierungsrelation über Aktivitäten
$\leq_T$	Spezialisierungsrelation über normsprachlichen Sätzen bzgl. Terminologie T
$\mathcal{S}_{Akt}(\cdot)$	Fachliche Semantik einer Aktivität eines Geschäftsprozessmodells
$\mathcal{S}_{Op}(\cdot)$	Fachliche Semantik einer Operation einer Komponente
$Impl(\cdot,\cdot)$	Implementierungsrelation zwischen Aktionen (Operationen und Aktivitäten)

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Literaturverzeichnis

[AB02] Aalst, W. M. P. van der und T. Basten: *Inheritance of Workflows: An Approach to Tackling Problems Related to Change.* Theoretical Computer Science, 270(1–2):125–203, 2002.

[ABC+02] Ackermann, J., F. Brinkop, S. Conrad, P. Fettke, A. Frick, E. Glistau, H. Jaekel, O. Kotlar, P. Loos, H. Mrech, E. Ortner, U. Raape, S. Overhage, S. Sahm, A. Schmietendorf, T. Teschke und K. Turowski: *Vereinheitlichte Spezifikation von Fachkomponenten*, Februar 2002. Memorandum des Arbeitskreises 5.10.3 *Komponentenorientierte betriebliche Anwendungssysteme.*

[ACD+03] Andrews, T., F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic und S. Weerawarana: *Business Process Execution Language for Web Services, Version 1.1.* BEA, IBM, Microsoft, SAP, Siebel Systems, 2003. Quelle: ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf.

[Ack01] Ackermann, J.: *Fallstudie zur Spezifikation von Fachkomponenten.* In: Turowski, K. (Herausgeber): *Tagungsband zum 2. Workshop „Modellierung und Spezifikation von Fachkomponenten" im Rahmen der vertIS (verteilte Informationssysteme auf der Grundlage von Objekten, Komponenten und Agenten) 2001*, Seiten 1–66, Bamberg, Oktober 2001.

[ADH+03] Aalst, W. M. P. van der, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm und A. J. M. M. Weijters: *Workflow Mining: A Survey of Issues and Approaches.* Data and Knowledge Engineering, 47(2):237–267, November 2003.

[AG97] Allen, R. und D. Garlan: *A Formal Basis for Architectural Connection.* ACM Transactions on Software Engineering and Methodology, 6(3):213–249, Juli 1997.

[AHT02] Aalst, W. M. P. van der, K. M. van Hee und R. A. van der Toorn: *Component-based Software Architectures: A Framework Based on Inheritance of Behavior.* Science of Computer Programming, 42(2–3):129–171, 2002.

[Ame91] America, P.: *Designing an Object-Oriented Programming Language with Behavioural Subtyping.* In: Bakker, J. W. de, W. P. de Roever und G. Rozenberg (Herausgeber): *Foundations of Object-Oriented Languages, Proceedings of REX School/Workshop, Noordwijkerhout, Niederlande, 28. Mai - 1. Juni 1990*, Band 489 der Reihe *Lecture Notes in Computer Science*, Seiten 60–90. Springer-Verlag, 1991.

[AO94] Apt, K. R. und E.-R. Olderog: *Programmverifikation.* Springer-Verlag, 1994.

[AR00] Appelrath, H.-J. und J. Ritter: *R/3-Einführung: Methoden und Werkzeuge.* SAP Kompetent. Springer-Verlag, 2000.

[ATA02] Air Transport Association: *AECMA Simplified English Guide for the Preparation of Aircraft Maintenance Documentation*, ATA publications 2002 CD-ROM, Issue 1, 2002.

[ATT03] AT&T Labs-Research, *Grappa – A Java Graph Package*. Elektronische Quelle: *http://www.research.att.com/~john/Grappa/*, Stand: September 2003.

[Bal00] Balzert, H.: *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, 2. Auflage, 2000.

[BCR00a] Börger, E., A. Cavarra und E. Riccobene: *An ASM Semantics for UML Activity Diagrams*. In: Rus, T. [Rus00], Seiten 293–308.

[BCR00b] Börger, E., A. Cavarra und E. Riccobene: *Modeling the Dynamics of UML State Machines*. In: Gurevich, Y., P. W. Kutter, M. Odersky und L. Thiele (Herausgeber): *Abstract State Machines*, Band 1912 der Reihe *Lecture Notes in Computer Science*, Seiten 223–241. Springer-Verlag, 2000.

[Ber02] Bergsten, H.: *JavaServer Pages*. O'Reilly & Associates, 2. Auflage, 2002.

[BGGM01] Batini, C., F. Giunchiglia, P. Giorgini und M. Mecella (Herausgeber): *Cooperative Information Systems, Proceedings of 9th International Conference on Cooperative Information Systems (CoopIS 2001), Trento, Italien, 5.-7. September 2001*, Band 2172 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[BGMV97] Borgo, S., N. Guarino, C. Masolo und G. Vetere: *Using a Large Linguistic Ontology for Internet-Based Retrieval of Object-Oriented Components*. In: *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering (SEKE '97)*, Madrid, Spanien, Juni 1997.

[BHR84] Brookes, S. D., C. A. R. Hoare und A. W. Roscoe: *A Theory of Communicating Sequential Processes*. Journal of the ACM, 31(3), 1984.

[Bie02] Bien, A.: *J2EE Patterns – Entwurfsmuster für die J2EE*. Addison-Wesley, 2002.

[Boo94] Booch, G.: *Object-Oriented Analysis and Design with Applications*. Benjamin/ Cummings Publishing Company, 2. Auflage, 1994.

[BP89] Biggerstaff, T. J und A. J. Perlis (Herausgeber): *Software Reusability, Band 1 – Concepts and Models*. Frontier Series. ACM Press, 1989.

[BR89] Biggerstaff, T. J. und C. Richter: *Reusability Framework, Assessment, and Directions*. In: Biggerstaff, T. J und A. J. Perlis [BP89], Seiten 1–17. Ebenfalls erschienen in IEEE Software, 13(2), März 1987.

[Bro95] Brockschmidt, K.: *Inside OLE*. Microsoft Press, 2. Auflage, 1995.

[Bro97] Broy, M.: *Towards a Mathematical Concept of a Component and its Use*. Software – Concepts & Tools, 18(3):137–148, 1997. Überarbeitete Version des gleichnamigen Beitrags zur *First Components User Conference*, München, 1996.

[BRS95] Becker, J., M. Rosemann und R. Schütte: *Grundsätze ordnungsmäßiger Modellierung*. Wirtschaftsinformatik, 37(5):435–445, Oktober 1995.

[BS86] Brinksma, E. und G. Scollo: *Formal Notions of Implementation and Conformance in LOTOS*. Technischer Bericht INF-86-13, Twente University of Technology, Department of Informatics, 1986.

[Büt95] Büttemeyer, W.: *Wissenschaftstheorie für Informatiker*. Spektrum Akademischer Verlag, 1995.

[Bui01] Buitelaar, P.: *Semantic Lexicons: Between Ontology and Terminology.* In: *Proceedings of OntoLex 2000: Ontologies and Lexical Knowledge Bases*, OntoText Lab., Sofia, Bulgarien, 2001.

[BW90] Baeten, J. C. M. und W. P. Weijland: *Process Algebra.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[CFTV00] Canal, C., L. Fuentes, J. M. Troya und A. Vallecillo: *Extending CORBA Interfaces with π-Calculus for Protocol Compatibility.* In: *Proceedings of TOOLS Europe 2000.* IEEE Press, Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga 2000.

[Cha96] Chappell, D.: *Understanding ActiveX and OLE.* Microsoft Press, 1996.

[CJB99] Chandrasekaran, B., J. R. Josephson und V. R. Benjamins: *What Are Ontologies, and Why Do We Need Them?* IEEE Intelligent Systems, 14(1):20–26, 1999.

[Cob01] Cobb, E.E.: *The Evolution of Distributed Component Architectures.* In: Batini, C. et al. [BGGM01], Seiten 7–21.

[Com03] ComponentSource, *ComponentSource: The Definitive Source of Software Components.* Elektronische Quelle: *http://www.componentsource.com/*, Stand: Mai 2003.

[CPT99] Canal, C., E. Pimentel und J. M. Troya: *Conformance and Refinement of Behavior in π-Calculus.* In: *Proceedings of Component-Based Development in Computational Logic*, Paris, September 1999.

[CT01] Conrad, S. und K. Turowski: *Temporal OCL: Meeting Specification Demands for Business Components.* In: Siau, K. und T. Halpin (Herausgeber): *Unified Modeling Language: Systems Analysis, Design and Development Issues*, Seiten 151–165. IDEA Group Publishing, 2001.

[DBSB91] Devanbu, P., R. J. Brachman, P. G. Selfridge und B. W. Ballard: *LaSSIE: A Knowledge-Based Software Information System.* Communications of the ACM, 34(5):35–49, Mai 1991.

[DDH72] Dahl, O.-J., E. W. Dijkstra und C. A. R. Hoare: *Structured Programming.* Academic Press, 1972.

[DG98] Deiters, W. und V. Gruhn: *Process Management in Practice: Applying the FUNSOFT Net Approach to Large-Scale Processes.* Automated Software Engineering, 5(1):7–25, 1998.

[DNH84] De Nicola, R. und M. Hennessy: *Testing equivalences for processes.* Theoretical Computer Science, 34:83–133, 1984.

[DW98] D'Souza, D. und A. Wills: *Objects, Components and Frameworks With UML: The Catalysis Approach.* Addison-Wesley, 1998.

[DYK01] DeMichiel, L. G., L. Ü. Yalçinalp und S. Krishnan: *Enterprise JavaBeans Specification, Version 2.0.* Sun Microsystems, 2001.

[EE94] Ebert, J. und G. Engels: *Observable or Invocable Behaviour – You Have to Choose!* Technischer Bericht tr94-38, Universität Koblenz, University of Leiden, 1994.

[EFLR98] Evans, A., R. France, K. Lano und B. Rumpe: *Developing the UML as a Formal Modelling Notation.* In: Bézivin, J. und P.-A. Muller (Herausgeber): *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop*, Seiten 297–307, Mulhouse, Frankreich, Juni 1998.

[FCF93] Fox, M., J.F. Chionglo und F.G. Fadel: *A Common Sense Model of the Enterprise*. In: *2nd Industrial Engineering Research Conference*, Seiten 425–429, Norcross GA: Institute for Industrial Engineers, 1993.

[Fis01] Fischer, B.: *Deduction-Based Software Component Retrieval*. Dissertation, Universität Passau, Fakultät für Mathematik und Informatik, November 2001.

[FS93] Ferstl, O. K. und E. J. Sinz: *Geschäftsprozeßmodellierung*. Wirtschaftsinformatik, 35(6):589–592, Dezember 1993.

[FS96a] Ferstl, O. K. und E. J. Sinz: *Geschäftsprozeßmodellierung im Rahmen des Semantischen Objektmodells*. In: Vossen, G. und J. Becker [VB96], Seiten 47–61.

[FS96b] Fuchs, N. E. und R. Schwitter: *Attempto Controlled English (ACE)*. In: *The First International Workshop on Controlled Language Applications (CLAW '96)*, Katholieke Universiteit Leuven, 1996.

[FS97] Fayad, M. E. und D. C. Schmidt: *Object-Oriented Application Frameworks*. Communications of the ACM, 40(10):33–38, Oktober 1997.

[FSS98] Fischer, B., J. Schumann und G. Snelting: *Deduction-Based Software Component Retrieval*. In: Bibel, W. und P. H. Schmitt (Herausgeber): *Automated Deduction – A basis for applications*, Band III: Applications, Seiten 265–292. Kluwer, 1998.

[FW00] Fischer, C. und H. Wehrheim: *Behavioural Subtyping Relations for Object-Oriented Formalisms*. In: Rus, T. [Rus00], Seiten 469–483.

[Gad02] Gadatsch, A.: *Management von Geschäftsprozessen – Methoden und Werkzeuge für die IT-Praxis: eine Einführung für Studenten und Praktiker*. Vieweg, Braunschweig, Wiesbaden, 2. Auflage, 2002.

[GG95] Guarino, N. und P. Giaretta: *Ontologies and Knowledge Bases - Towards a Terminological Clarification*. In: Mars, N.J. (Herausgeber): *Towards Very Large Knowledge Bases - Knowledge Building and Knowledge Sharing 1995*, Seiten 25–32. IOS Press, Amsterdam, 1995.

[GHJV01] Gamma, E., R. Helm, R. Johnson und J. Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1. Auflage, 2001.

[GI94a] Girardi, M. R. und B. Ibrahim: *Automatic Indexing of Software Artifacts*. In: Frakes, W. B. (Herausgeber): *3rd International Conference on Software Reusability*, Rio de Janeiro, Brazil, 1994. IEEE Computer Society Press, Los Alamitos.

[GI94b] Girardi, M. R. und B. Ibrahim: *A Similarity Measure for Retrieving Software Artifacts*. In: *6th International Conference on Software Engineering and Knowledge Engineering*, Jurmala, Lettland, 1994.

[Gla90] Glabbeek, R. J. van: *The linear time - branching time spectrum*. In: Baeten, J. C. M. und J. W. Klop (Herausgeber): *CONCUR 90*, Band 458 der Reihe *Lecture Notes in Computer Science*, Seiten 278–297. Springer-Verlag, 1990.

[Gla93] Glabbeek, R. J. van: *The linear time - branching time spectrum II: The semantics of sequential systems with silent moves*. In: Best, E. (Herausgeber): *CONCUR 93*, Band 715 der Reihe *Lecture Notes in Computer Science*, Seiten 66–81. Springer-Verlag, 1993.

[GMV99] Guarino, N., C. Masolo und G. Vetere: *OntoSeek: Content-Based Access to the Web*. IEEE Intelligent Systems, 14(3):70–80, Mai/Juni 1999.

[GMW00] Garlan, D., R. T. Monroe und D. Wile: *Acme: Architectural Description of Component-Based Systems*. In: Leavens, G. T. und M. Sitaraman [LS00], Seiten 47–67.

[GN87] Genesereth, M. R. und N. J. Nilsson: *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Los Altos, 1987.

[Gri98] Griffel, F.: *Componentware*. dpunkt, Heidelberg, 1998.

[Gro03] Gronau, N.: *Wandlungsfähige Informationssystemarchitekturen – Nachhaltigkeit bei organisatorischem Wandel*. GITO-Verlag, Berlin, 2003.

[Gru93] Gruber, T. R.: *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*. In: Guarino, N. und R. Poli (Herausgeber): *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, Niederlande, 1993. Kluwer Academic Publishers.

[GSB+02] Graham, S., S. Simeonov, T. Boubez, D. Davis, G. Daniels, Y. Nakamura und R. Neyama: *Building Web Services with Java – Making Sense of XML, SOAP, WSDL, and UDDI*. Sams Publishing, 2002.

[GT00] Gruhn, V. und A. Thiel: *Komponentenmodelle – DCOM, JavaBeans, Enterprise JavaBeans, CORBA*. Professionelle Softwareentwicklung. Addison-Wesley, 2000.

[GV90] Groote, J. F. und F. Vaandrager: *An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence*. In: Paterson, M. (Herausgeber): *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, 16.-20. Juli 1990*, Band 443 der Reihe *Lecture Notes in Computer Science*, Seiten 626–638. Springer-Verlag, 1990.

[GW96] Glabbeek, R. J. van und W. P. Weijland: *Branching time and abstraction in bisimulation semantics*. Journal of the ACM, 43(3):555–600, 1996.

[Har90] Hartmann, D.: *Konstruktive Fragelogik – Vom Elementarsatz zur Logik von Frage und Antwort*. BI-Wissenschaftsverlag, Mannheim, Wien, Zürich, 1990.

[Has02] Hasselbring, W.: *Component-Based Software Engineering*. In: Chang, S.K. (Herausgeber): *Handbook of Software Engineering and Knowledge Engineering*, Band 2. World Scientific Publishing, 2002.

[Hau01] Hau, M.: *Das DATEV-Komponenten-Repository – Ein Beitrag zu Marktplätzen für betriebswirtschaftliche Software-Bausteine*. FORWIN-Bericht FWN-2001-003, FORWIN – Bayerischer Forschungsverbund Wirtschaftsinformatik, 2001.

[HC94] Hammer, M. und J. Champy: *Business Reengineering – Die Radikalkur für das Unternehmen*. Campus, Frankfurt/Main, New York, 1994.

[Hen94] Henninger, S.: *Using Iterative Refinement to Find Reusable Software*. IEEE Software, 11(5):48–59, September/Oktober 1994.

[Heu02] Heuvel, W.-J. van den: *Integrating Modern Business Applications with Objectified Legacy Systems*. Ph.D. Thesis, Tilburg University, Faculty of Economics and Business Administration, 2002.

[HG97] Harel, D. und E. Gery: *Executable Object Modeling with Statecharts*. IEEE Computer, 30(7), 1997.

[HK00] Harel, D. und O. Kupferman: *On the Behavioral Inheritance of State-Based Objects*. In: *Proc. 34th Int. Conf. on Component and Object Technology*, Santa Barbara, CA, August 2000. IEEE Computer Society Press, Los Alamitos.

[HK03] Hitz, M. und G. Kappel: *UML@Work*. dpunkt.verlag, 2., aktualisierte und überarbeitete Auflage, 2003.

[HM02] Hau, M. und P. Mertens: *Computergestützte Auswahl komponentenbasierter Anwendungssysteme*. Informatik-Spektrum, 25(5):331–340, Oktober 2002.

[Hoa69] Hoare, C. A. R.: *An axiomatic basis for computer programming*. Communications of the ACM, 12(10):576–583, Oktober 1969.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice Hall, New York, 1985.

[HS96] Hüttel, H. und S. Shukla: *On the Complexity of Deciding Behavioural Equivalences and Preorders – A Survey*. Basic Research in Computer Science (BRICS) RS-96-39, Department of Computer Science, University of Aarhus, 1996.

[HS00] Herzum, P. und O. Sims: *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley & Sons, 2000.

[HT01] Harren, A. und H. Tapken: *TODAY Open Repository: An Extensible, MOF-Based Metadata Repository System*. Technischer Bericht, OFFIS, Oldenburg, 2001.

[Hun01] Hunter, J.: *Java Servlet Programming*. O'Reilly & Associates, 2. Auflage, 2001.

[HYP01] Heuvel, W.-J. van den, J. Yang und M. P. Papazoglou: *Service Representation, Discovery, and Composition for E-marketplaces*. In: Batini, C. et al. [BGGM01], Seiten 270–284.

[JCJO92] Jacobson, I., M. Christerson, P. Jonsson und G. Övergaard: *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, 1992.

[JM97] Jézéquel, J.-M. und B. Meyer: *Design by Contract: The Lessons of Ariane*. IEEE Computer, 30(1):129–130, 1997.

[Joh97] Johnson, R. E.: *Frameworks = (Components + Patterns)*. Communications of the ACM, 40(10):39–42, Oktober 1997.

[JT02] Jaekel, H. und T. Teschke: *Berücksichtigung von Berechtigungskonzepten bei der Spezifikation von Fachkomponenten*. In: Turowski, K. (Herausgeber): *Tagungsband zum 3. Workshop „Modellierung und Spezifikation von Fachkomponenten" im Rahmen der Multi-Konferenz Wirtschaftsinformatik 2002 (MK-WI '02)*, Seiten 69–85, Nürnberg, 2002.

[KAMN98] Kamprath, C., E. Adolphson, T. Mitamura und E. Nyberg: *Controlled Language for Multilingual Document Production: Experience with Caterpillar Technical English*. In: *Proceedings of the Second International Workshop of Controlled Language Applications (CLAW '98)*, Pittsburgh, USA, Mai 1998.

[KAO03] KAON, *KAON – The Karlsruhe Ontology and Semantic Web Tool Suite*. Elektronische Quelle: *http://kaon.semanticweb.org/*, Stand: September 2003.

[Kar95] Karlsson, E.-A. (Herausgeber): *Software Reuse, A Holistic Approach*. John Wiley & Sons, 1995.

[Kau00a] Kaufmann, T.: *Entwurf eines Marktplatzes für heterogene Komponenten betrieblicher Anwendungssysteme*. Dissertation, Universität Erlangen-Nürnberg, Bereich Wirtschaftsinformatik I, 2000.

[Kau00b] Kaufmann, T.: *Marktplatz für Bausteine heterogener betrieblicher Anwendungssysteme.* FORWIN-Bericht FWN-2000-003, FORWIN – Bayerischer Forschungsverbund Wirtschaftsinformatik, 2000.

[Kei01] Keilers, M.: *Ein internetbasierter Komponentensuchdienst auf Basis von Verhaltensbeschreibungen.* Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Fachbereich Informatik, 2001.

[KL94] Knight, K. und S. Luk: *Building a Large Knowledge Base for Machine Translation.* In: *Proceedings of the 12th American Association of Artificial Intelligence Conference (AAAI '94)*, Seattle, USA, Juli 1994.

[KLM01] Kaufmann, T., M. Lohmann und P. Morschheuser: *Die Informationsdatenbank ICF – eine wissensbasierte Werkzeugsammlung für die Software-Anforderungsanalyse.* FORWIN-Bericht FWN-2001-002, FORWIN – Bayerischer Forschungsverbund Wirtschaftsinformatik, 2001.

[KNS92] Keller, G., M. Nüttgens und A.-W. Scheer: *Semantische Prozeßmodellierung auf der Grundlage „Ereignisgesteuerter Prozeßketten (EPK)“.* Technischer Bericht 89, Institut für Wirtschaftsinformatik, Universität des Saarlandes, 1992.

[Krä98] Krämer, B.: *Synchronization Constraints in Object Interfaces.* In: Krämer, B., M. P. Papazoglou und H. W. Schmidt (Herausgeber): *Information Systems Interoperability*, Seiten 111–141. Research Studies Press, Taunton, England, 1998.

[KRH03] Kratz, B., R. H. Reussner und W.-J. van den Heuvel: *Empirical Research on Similarity Metrics for Software Component Interfaces.* In: B. Hua et al. (Herausgeber): *Proceedings of Integrated Design and Process Technology (IDPT 2003), zur Veröffentlichung angenommen*, Peking, China, 2003. Society for Design and Process Science (SDPS).

[Kru92] Krueger, C. W.: *Software Reuse.* ACM Computing Surveys, 24(2):131–183, Juni 1992.

[KT97] Keller, G. und T. Teufel: *SAP R/3 prozeßorientiert anwenden – Iteratives Prozeß-Prototyping zur Bildung von Wertschöpfungsketten.* Edition SAP. Addison-Wesley, 1997.

[LBR99] Leavens, G. T., A. L. Baker und C. Ruby: *Preliminary Design of JML: A Behavioural Interface Specification Language for Java.* Technical Report 98-06g, Iowa State University, Department of Computer Science, August 1999.

[LD00] Leavens, G. T. und K. K. Dhara: *Concepts of Behavioural Subtyping and a Sketch of Their Extension to Component-Based Systems.* In: Leavens, G. T. und M. Sitaraman [LS00], Seiten 111–135.

[LG99] Lawrence, S. und C. L. Giles: *Accessibility of Information on the Web.* Nature, 400:107–109, 1999.

[Lor87] Lorenzen, P.: *Lehrbuch der konstruktiven Wissenschaftstheorie.* BI-Wissenschaftsverlag, Mannheim, Wien, Zürich, 1987.

[LS00] Leavens, G. T. und M. Sitaraman (Herausgeber): *Foundations of Component-Based Systems.* Cambrigde University Press, 2000.

[LW94] Liskov, B. und J. Wing: *A Behavioral Notion of Subtyping.* ACM Transactions on Programming Languages and Systems, 16(6):1811–1841, 1994.

[Mat96] Mattsson, M.: *Object-Oriented Frameworks: A Survey of Methodological Issues*. Technical Report 96-167, University of Karlskrona/Ronneby, Department of Software Engineering and Computer Science, 1996.

[McI69] McIlroy, M. D.: *"Mass Produced" Software Components*. In: Naur, P. und B. Randell (Herausgeber): *Software Engineering*, Seiten 138–155, Brüssel, Belgien, 1969. Scientific Affairs Division, NATO. Bericht zu einer durch das NATO Science Committee gesponsorten Konferenz, Garmisch (Deutschland), 7.-11. Oktober 1968.

[MDC99] Meta Data Coalition: *Open Information Model, Version 1.1 (Proposal)*, August 1999.

[MDEK95] Magee, J., N. Dulay, S. Eisenbach und J. Kramer: *Specifying Distributed Software Architectures*. In: *5th European Software Engineering Conference (ESEC '95)*, Barcelona, 1995.

[Mey91] Meyer, B.: *Eiffel: The Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Mey92] Meyer, B.: *Applying "Design By Contract"*. IEEE Computer, 25(10):40–51, 1992.

[Mil71] Milner, R.: *An Algebraic Definition of Simulation between Programs*. In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, Seiten 481–489, London, 1971. British Computer Society.

[Mil80] Milner, R.: *A Calculus of Communicating Systems*, Band 92 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil89] Milner, R.: *Communication and Concurrency*. Prentice Hall, New York, 1989.

[Mil95] Miller, G. A.: *WordNet: A Lexical Database for English*. Communications of the ACM, 38(11):39–41, November 1995.

[MMM94] Mili, A., R. Mili und R. T. Mittermeir: *Storing and Retrieving Software Components: A Refinement Based System*. In: *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*, Seiten 91–100, Sorrento, Italien, Mai 1994. IEEE Computer Society / ACM Press. Eine überarbeitete Version ist in IEEE Transactions on Software Engineering, 23(7):445–460, 1997 erschienen.

[MMM98] Mili, A., R. Mili und R. T. Mittermeir: *A survey of software reuse libraries*. Annals of Software Engineering, 5:349–414, 1998.

[MMV02] Motik, B., A. Maedche und R. Volz: *A Conceptual Modeling Approach for Building Semantics-Driven Enterprise Applications*. In: Meersman, R. und Z. Tari (Herausgeber): *On the Move to Meaningful Internet Systems, 2002 – DOA/CoopIS/ ODBASE, Proceedings of the Confederated International Conferences DOA, CoopIS and ODBASE 2002*, Band 2519 der Reihe *Lecture Notes in Computer Science*, Seiten 1082–1099, Irvine, USA, 2002. Springer-Verlag.

[Moh03] Mohan, C., *Tutorial: Application Servers and Associated Technologies*. Elektronische Quelle: *http://www.almaden.ibm.com/u/mohan/AppServersTutorial_ SIGMOD2002_ Slides.pdf*, Stand: September 2003.

[MPW92] Milner, R., J. Parrow und D. Walker: *A Calculus of Mobile Processes, Part I and II*. Journal of Information and Computation, 100(1):1–77, 1992.

[MSS01] Maedche, A., S. Staab und R. Studer: *Ontologien*. Wirtschaftsinformatik, 43(4):393–395, August 2001.

[ND95] Nierstrasz, O. und L. Dami: *Component-Oriented Software Technology.* In: Nierstrasz, O. und D. Tsichritzis [NT95], Seiten 3–28.

[Nie95] Nierstrasz, O.: *Regular Types for Active Objects.* In: Nierstrasz, O. und D. Tsichritzis [NT95], Seiten 99–121. Überarbeitete und korrigierte Version des gleichnamigen Beitrags zur *8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93)*, Seiten 1–15, Oktober 1993, Washington DC, USA.

[Nom03] Nomina, *Software-Marktplatz: Die ISIS Software- und Firmendatenbanken.* Elektronische Quelle: *http://www.software-marktplatz.de/*, Stand: Mai 2003.

[NT95] Nierstrasz, O. und D. Tsichritzis (Herausgeber): *Object-Oriented Software Composition.* Prentice Hall, 1995.

[Oes98] Oestereich, B.: *Objektorientierte Geschäftsprozessmodellierung mit der UML.* OBJEKTspektrum, 2:48–52, 1998.

[Oes01] Oestereich, B.: *Objektorientierte Softwareentwicklung: Analyse und Design mit der UML.* R. Oldenbourg Verlag, 5. Auflage, 2001.

[OHE96] Orfali, R., D. Harkey und J. Edwards: *The Essential Distributed Objects Survival Guide.* John Wiley & Sons, New York, 1996.

[OHE99] Orfali, R., D. Harkey und J. Edwards: *Client/Server Survival Guide.* John Wiley & Sons, New York, 3. Auflage, 1999.

[OL82] Owicki, S. und L. Lamport: *Proving Liveness Properties of Concurrent Programs.* ACM Transactions on Programming Languages and Systems (TOPLAS), 4(3):455–495, 1982.

[Old91] Olderog, E.-R.: *Nets, Terms and Formulas.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1991.

[OLK99] Ortner, E., K.-P. Lang und J. Kalkmann: *Anwendungssystementwicklung mit Komponenten.* Information Management & Consulting, 14(2):35–45, 1999.

[OMG98] Object Management Group: *Business Object Component Architecture (BOCA), Proposal, Revision 1.1, OMG document bom/98-01-07*, 1998.

[OMG00a] Object Management Group: *Meta Object Facility (MOF) Specification, Version 1.3*, März 2000.

[OMG00b] Object Management Group: *XML Metadata Interchange (XMI) Specification, Version 1.1*, November 2000.

[OMG02a] Object Management Group: *Common Object Request Broker Architecture: Core Specification, Version 3.0*, Dezember 2002.

[OMG02b] Object Management Group: *UML Profile for Enterprise Distributed Object Computing Specification, OMG Adopted Specification, Dokument ptc/02-02-05*, Februar 2002.

[OMG03] Object Management Group: *Unified Modeling Language Specification, Version 1.5, OMG document formal/03-03-01*, März 2003.

[Ort97] Ortner, E.: *Methodenneutraler Fachentwurf.* Wirtschaftsinformatik. B. G. Teubner Verlag, 1997.

[Par81] Park, D. M. R.: *Concurrency and Automata for Infinite Sequences.* In: *Proceedings of 5th GI Conference*, Band 104 der Reihe *Lecture Notes in Computer Science.* Springer-Verlag, 1981.

[PDF87] Prieto-Diáz, R. und P. Freeman: *Classifying Software for Reusability.* IEEE Software, 4(1):6–16, 1987.

[Plo81] Plotkin, G. D.: *Structured Approach to Operational Semantics.* Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.

[Poz01] Pozewaunig, H.: *Mining Component Behaviour to Support Software Retrieval.* Dissertation, Universität Klagenfurt, Fakultät für Wirtschaftswissenschaften und Informatik, Oktober 2001.

[PP93] Podgurski, A. und L. Pierce: *Retrieving Reusable Software by Sampling Behaviour.* ACM Transactions on Software Engineering and Methodology, 2(3):286–303, Juli 1993.

[Pro02] Prosie, J.: *.NET Entwicklerbuch.* Microsoft Press, 2002.

[PS84] Patel-Schneider, P. F.: *Small can be beautiful in knowledge representation.* In: *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, Seiten 11–16, Denver, Colorado, USA, 1984.

[PUM03] The precise UML group, *Home page.* Elektronische Quelle: *http://www.puml.org/*, Stand: Juli 2003.

[Pun96] Puntigam, F.: *Types for Active Objects Based on Trace Semantics.* In: *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS '96)*, Paris, Frankreich, 1996.

[RBP+91] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy und W. Lorensen: *Object Oriented Modeling and Design.* Prentice-Hall, 1991.

[Rei85] Reisig, W.: *Petri Nets: An Introduction*, Band 4 der Reihe *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[Rit98] Ritter, J.: *PROSECCO – Eine Methode zur modellbasierten Anwendungssystemgestaltung.* In: *Proceedings of Business Information Systems '98*, Posen, Polen, 1998.

[Rit00] Ritter, J.: *Prozessorientierte Konfiguration komponentenbasierter Anwendungssysteme.* Dissertation, Carl von Ossietzky Universität Oldenburg, Fachbereich Informatik, 2000.

[Ros96] Rosemann, M.: *Komplexitätsmanagement in Prozeßmodellen - Methodenspezifische Gestaltungsempfehlungen für die Informationsmodellierung.* Schriften zur EDV-orientierten Betriebswirtschaftslehre. Gabler, Wiesbaden, 1996.

[Ros97] Roscoe, A. W.: *The Theory and Practice of Concurrency.* Prentice Hall Series in Computer Science. Prentice Hall, 1997.

[Rum99] Rump, F. J.: *Geschäftsprozeßmanagement auf der Basis ereignisgesteuerter Prozeßketten – Formalisierung, Analyse und Ausführung von EPKs.* Reihe Wirtschaftsinformatik. B. G. Teubner Verlag, 1999.

[Rus00] Rus, T. (Herausgeber): *8th International Conference on Algebraic Methodology and Software Technology (AMAST 2000)*, Band 1816 der Reihe *Lecture Notes in Computer Science*, Iowa City, Iowa, USA, Mai 2000. Springer-Verlag.

[Sam97] Sametinger, J.: *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[SAP03] SAP, *SAP – Software Partner Directory*. Elektronische Quelle: *http://www.mysap.com/partners/software/directory/*, Stand: Mai 2003.

[Saw01] Sawyer, S.: *A Market-Based Perspective on Information Systems Development*. Communications of the ACM, 44(11):97–102, November 2001.

[Sch92] Schöning, U.: *Theoretische Informatik kurz gefaßt*. BI Wissenschaftsverlag, 1992.

[Sch97] Schienmann, B.: *Objektorientierter Fachentwurf*, Band 20 der Reihe *Teubner-Texte zur Informatik*. B. G. Teubner Verlag, 1997.

[Sch98a] Scheer, A.-W.: *ARIS – Modellierungsmethoden, Metamodelle, Anwendungen*. Springer-Verlag, 3. Auflage, 1998.

[Sch98b] Scheer, A.-W.: *ARIS – Vom Geschäftsprozeß zum Anwendungssystem*. Springer-Verlag, 3. Auflage, 1998.

[Sch01] Schimm, G.: *Process Mining linearer Prozessmodelle - Ein Ansatz zur automatisierten Akquisition von Prozesswissen*. In: *Tagungsband 1. Konferenz Professionelles Wissensmanagement*, Baden-Baden, 2001.

[Sch02] Schimm, G.: *Process Miner - A Tool for Mining Process Schemes from Event-based Data*. In: Flesca, S., S. Greco, N. Leone und G. Ianni (Herausgeber): *Proceedings of the 8th European Conference on Artificial Intelligence*, Band 2424 der Reihe *Lecture Notes in Artificial Intelligence*, Seiten 525–528. Springer-Verlag, 2002.

[Sch03] Schimm, G.: *Mining Most Specific Workflow Models from Event-based Data*. In: Aalst, W. M. P. van der, A. ter Hofstede und M. Weske (Herausgeber): *Business Process Management, Proceedings of the International Conference BPM 2003*, Band 2678 der Reihe *Lecture Notes in Computer Science*, Seiten 25–40, Eindhoven, Niederlande, Juni 2003. Springer-Verlag.

[SG96] Shaw, M. und D. Garlan: *Software Architecture – Perspectives on an Emerging Discipline*. Prentice-Hall, Upper Saddle River, New Jersey, 1996.

[SGT+00] Schlenoff, C., M. Gruninger, F. Tissot, J. Valois, J. Lubell und J. Lee: *The Process Specification Language (PSL): Overview and Version 1.0 Specification*. Technischer Bericht NISTIR 6459, National Institute of Standards and Technology (NIST), 2000.

[SHJ+94] Saake, G., P. Hartel, R. Jungclaus, R. Wieringa und R. Feenstra: *Inheritance Conditions for Object Life Cycle Diagrams*. In: Lipeck, U. und G. Vossen (Herausgeber): *Workshop Formale Grundlagen für den Entwurf von Informationssystemen*, Seiten 89–95, Tutzing, 1994.

[SM83] Salton, G. und M. J. McGill: *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[Sne52] Snell, B.: *Der Aufbau der Sprache*. Claassen Verlag, Hamburg, 1952.

[SNZ97] Scheer, A.-W., M. Nüttgens und V. Zimmermann: *Objektorientierte Ereignisgesteuerte Prozeßkette (oEPK) – Methode und Anwendung*. Technischer Bericht 141, Institut für Wirtschaftsinformatik, Universität des Saarlandes, 1997.

[Sof03] SoftSelect, *SoftSelect Matching Plattform*. Elektronische Quelle: *http://www.softselect.de/*, Stand: Mai 2003.

[Sou03] SourceForge.net, *Open Source Software Development Website*. Elektronische Quelle: *http://sourceforge.net/*, Stand: Mai 2003.

[Spr00] Sprott, D.: *Componentizing the Enterprise Application Packages*. Communications of the ACM, 43(4):63–69, April 2000.

[SSSS01] Staab, S., S. Studer, H. P. Schnurr und Y. Sure: *Knowledge Processes and Ontologies*. IEEE Intelligent Systems, 16(1):26–34, 2001.

[STK02] Snell, J., D. Tidwell und P. Kulchenko: *Programming Web Services with SOAP*. O'Reilly & Associates, 2002.

[STR00] Sandmann, C., T. Teschke und J. Ritter: *Ein Vorgehensmodell für die komponentenbasierte Anwendungsentwicklung*. In: Britzelmaier, B. und S. Geberl (Herausgeber): *Information als Erfolgsfaktor*, Seiten 49–58. B. G. Teubner Verlag, 2000.

[Sun03] Sun Microsystems, *iForce Partner Products Catalog*. Elektronische Quelle: *http://solutions.sun.com/direct/portfolio.jsp?msg=J2P*, Stand: Mai 2003.

[Szy02] Szyperski, C. mit D. Gruntz und S. Murer: *Component Software - Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley, 2. Auflage, 2002.

[Tan90] Tanimoto, S. L.: *KI: Die Grundlagen*. Oldenbourg Verlag, 1990.

[Tes01] Teschke, T.: *Using Business Process Knowledge for Software Component Retrieval*. In: *Proceedings of 11th Annual BIT Conference*, Manchester, 2001.

[Tes02] Teschke, T.: *Ontological Intermediation between Business Process Models and Software Components*. In: Haav, H.-M. und A. Kalja (Herausgeber): *Databases and Information Systems II, Selected Papers from the Fifth International Baltic Conference, BalticDB&IS 2002, Tallinn, Estland*. Kluwer Academic Publishers, 2002.

[TR00] Teschke, T. und J. Ritter: *Towards a Foundation of Component-Oriented Software Reference Models*. In: Butler, G. und S. Jarzabek (Herausgeber): *Generative and Component-Based Software Engineering, Revised Papers of 2nd International Symposium GCSE 2000, Erfurt, Deutschland, 9. - 12. September 2000*, Band 2177 der Reihe *Lecture Notes in Computer Science*, Seiten 71–85. Springer-Verlag, 2000.

[Tra88] Tracz, W.: *Software Reuse Maxims*. ACM SIGSOFT Software Engineering Notes, 13(4):28–31, Oktober 1988.

[TUC03] Technische Universität Chemnitz, Informationssysteme & Management, *Component Markets – An Overview*. Elektronische Quelle: *http://www.tu-chemnitz.de/wirtschaft/wi2/projects/components/*, Stand: Mai 2003.

[UKMZ98] Uschold, M., M. King, S. Moralee und Y. Zorgios: *The Enterprise Ontology*. The Knowledge Engineering Review, Special Issue on Putting Ontologies to Use, 13(1), 1998.

[VB96] Vossen, G. und J. Becker (Herausgeber): *Geschäftsprozeßmodellierung und Workflow-Management – Modelle, Methoden, Werkzeuge*. International Thomson Publishing, Bonn, Albany, 1996.

[VHT99] Vallecillo, A., J. Hernández und J. M. Troya (Herausgeber): *Proceedings of the ECOOP '99 Workshop on Object Interoperability*, Lissabon, Portugal, Juni 1999.

[W3C99] World Wide Web Consortium (W3C): *Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation*, Februar 1999. Quelle: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[W3C02] World Wide Web Consortium (W3C): *Web Services Choreography Interface (WSCI) 1.0, W3C Note*, August 2002. Quelle: http://www.w3.org/TR/2002/NOTE-wsci-20020808.

[W3C03] World Wide Web Consortium (W3C): *Web Services Architecture, W3C Working Draft*, Mai 2003. Quelle: http://www.w3.org/TR/2003/WD-ws-arch-20030514.

[Wed92] Wedekind, H.: *Objektorientierte Schema-Entwicklung*, Band 85 der Reihe *Informatik*. BI-Wissenschaftsverlag, Mannheim, Wien, Zürich, 1992.

[Weh99] Wehrheim, H.: *Data Abstraction for CSP-OZ*. In: Wing, J. M., J. Woodcock und J. Davies (Herausgeber): *FM'99 - Formal Methods, World Congress on Formal Methods, Volume II*, Band 1709 der Reihe *Lecture Notes in Computer Science*, Seiten 1028–1047. Springer-Verlag, September 1999.

[Weh02] Wehrheim, H.: *Behavioural Subtyping in Object-Oriented Specification Formalisms*. Habilitationsschrift, Carl von Ossietzky Universität Oldenburg, Fachbereich Informatik, 2002.

[WO80] Wedekind, H. und E. Ortner: *Systematisches Konstruieren von Datenbankanwendungen: zur Methodologie der Angewandten Informatik*, Band 16 der Reihe *Applied Computer Science*. Carl Hanser Verlag, München, 1980.

[WWB+99] Winter, A., A. Winter, K. Becker, O. Bott, B. Brigl, S. Gräber, W. Hasselbring, R. Haux, C. Jostes, O.-S. Penger, H.-U. Prokosch, J. Ritter, R. Schütte und A. Terstappen: *Referenzmodelle für die Unterstützung des Managements von Krankenhausinformationssystemen*. Informatik, Biometrie und Epidemiologie in Medizin und Biologie, 30(4):173–189, 1999.

[WZ88] Wegner, P. und S. B. Zdonik: *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like*. In: Gjessing, S. und K. Nygaard (Herausgeber): *Proceedings ECOOP '88*, Band 322 der Reihe *Lecture Notes in Computer Science*, Seiten 55–77, Oslo, Norwegen, 1988. Springer-Verlag.

[ZW93] Zaremski, A. M. und J. M. Wing: *Signature Matching: A Key to Reuse*. Software Engineering Notes, 18(5):182–190, 1993.

[ZW97] Zaremski, A. M. und J. M. Wing: *Specification Matching of Software Components*. ACM Transactions on Software Engineering and Methodology, 6(4):333–369, 1997.

Index

Lebenslauf

Persönliche Daten

Name:	Thorsten Teschke
Wohnort:	Unter den Linden 47 26129 Oldenburg
Geburtsdatum:	03.06.1971
Geburtsort:	Bremerhaven
Familienstand:	ledig
Staatsangehörigkeit:	deutsch

Schulischer Bildungsweg

1978 – 1982	Grundschule Bexhövede
1982 – 1983	Haupt- und Realschule mit Orientierungsstufe Loxstedt
1983 – 1984	Haupt- und Realschule mit Orientierungsstufe Hohenkirchen
1984 – 1988	Progymnasium Hohenkirchen, späteres Gymnasium Wangerland
1988 – 1991	Mariengymnasium Jever
5/1991	Erlangung der Allgemeinen Hochschulreife (Abitur)

Wehrdienst

1991 – 1992	Grundwehrdienst als Stabssoldat beim Jagdbombergeschwader 38 „Friesland“ in Jever

Studium

1992 – 1998	Studium der Informatik mit Nebenfach Betriebswirtschaftslehre an der Carl von Ossietzky Universität Oldenburg
1998	Aufenthalt an der Carnegie Mellon University in Pittsburgh/ USA als Visiting Scholar
8/1998	Erlangung des akademischen Grads „Diplom-Informatiker“

Berufstätigkeit

seit 1998	Wissenschaftlicher Mitarbeiter am Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme (OFFIS)